

# A trace-based model for multi-party contracts\*

Tom Hvitved

Department of Computer Science, University of Copenhagen  
Universitetsparken 1, DK-2100 Copenhagen, Denmark  
hvitved@diku.dk

A (multi-party) contract is a legally binding agreement between individuals or companies that describes the commitments of each contract participant. For enterprises, contracts serve as the *external interface* to their clients, and consequently it is crucial to *monitor* the execution of these contracts for violations, and to *comply* with them in order to avoid financial penalties. In this paper we present a trace-based model for multi-party contracts, in which contract conformance is defined abstractly as a property on traces. A highlight of our model is *blame assignment*, which means that all contract violations are attributed to contract participants, and we analyze how blame assignment affects compositionality. Moreover, in order to specify contracts, we introduce a contract specification language, *CSL*, which is given a formal semantics by means of a mapping into the abstract model, and which overcomes the limitations of existing contract languages by supporting (a) (history sensitive and conditional) commitments, (b) parametrized contract templates, (c) relative and absolute temporal constraints, (d) contrary-to-duty obligations, (e) potentially infinite contracts, and (f) arithmetic expressions. Last but not least, *CSL* admits run-time monitoring via rewriting.

## 1 Introduction

In the recent years, fully automated *contract lifecycle management* (CLM) has become a critical key to success for enterprises, as identified by the Aberdeen Group [11]. However, current state-of-the-art enterprise systems such as Microsoft Dynamics NAV<sup>1</sup> do not represent contracts explicitly as first-class objects, but rather implicitly via low-level code and database schemes. The implications of such implicit encodings are that it is unclear what is implemented, and whether the implementation conforms with the actual contracts. Furthermore, changes are often costly and time consuming, and analysis of (running) contracts are difficult to perform. To overcome these limitations, various authors have proposed domain specific languages for representing contracts [1, 2, 3, 4, 6, 13], and also the before mentioned studies by the Aberdeen Group suggest using a domain specific language as the basis for automated CLM. However, constructing such domain specific language is a challenge, since contracts can involve different aspects, such as absolute temporal constraints (deadlines), relative temporal constraints (sequential ordering), contrary-to-duty clauses, conditional commitments, different deontic modalities and repetitive patterns. In light of these challenges, existing contract languages fall short on different important aspects, and even only few contract languages are presented with a formal semantics [1, 13]. There are several reasons why specification of contracts in a domain specific language is desirable, e.g., in order to analyze contracts prior to signing them, as described by Peyton-Jones and Eber [12] for financial contracts. But the most important reason to formalize contracts is the ability to *run-time monitor* their execution: a formalized contract should be *executable* with respect to the events relevant to the contract, and the execution (monitoring) should report violations by contract participants as they occur. The aspect of

---

\*Extended abstract. This work was done during the author's internship in the Information Security group at ETH Zürich.

<sup>1</sup><http://www.microsoft.com/dynamics/en/us/products/nav-overview.aspx>

*blame assignment* is a fundamental property that is not handled by any existing contract languages, even though run-time monitoring of contracts has been studied extensively [1, 3, 4, 8, 13, 14].

The motivation for our work is consequently the construction of a contract language which supports as many of the various features as possible, and which permits run-time monitoring with blame assignment. However, rather than beginning at the high level of a contract language and all the features it should support, we must first analyze what the abstract properties of contracts are — only when we have a *good* abstract mathematical model, which matches the intuition of contracts, can we consider a language. Only few authors have previously considered more abstract contract models [8, 9, 14], and among those only Xu [14] considers blame assignment. However, these models are not fully abstract, since they rely on for instance deadlines [14], deontic modalities [8] and logical formulae [9], and furthermore none of the authors have constructed contract languages for specifying contracts.

The approach we take is a *trace-based* model of contracts, which means that contracts are defined as properties on the history of actions that have taken place, i.e., contracts are *driven* by the actions performed (similar to Pace and Schneider [10] and Andersen et al. [1]). Depending on the sequence of actions that take place, the outcome of a contract can either be fulfillment (contract *conformance*) or non-fulfillment (contract *violation*), and such outcome may then for instance be *suggestive* of an obligation or a deadline not being met — but the model does not rely on such high-level notions. As argued above, the contract must also specify *who* violates the contract in case of nonfulfillment: in our model, contracts are restricted to *deterministic* blame assignment, which means that all violations can be uniquely attributed to contract participants. Even though deterministic blame assignment is certainly a desirable feature, not all contracts need be deterministic, but for now we restrict the model to deterministic blame assignment. Moreover, violations are considered *fundamental breaches*, which simplified means that a contract is invalid once it is violated, and we consequently do not model violations at different time points.

A desirable property of any formalism is *compositionality*, which for contracts means the ability to combine a set of sub contracts — typically called paragraphs or clauses — into one contract. A highlight of our work is an abstract account of contract conjunction and contract disjunction, and how these compositions affect blame assignment. Contract conjunction is evident in nearly all paper contracts — typically in the form of a set of implicitly conjuncted paragraphs — and we show that contract conjunction can be given a coherent interpretation. The composition relies on the before mentioned assumption of fundamental breaches: if a contract contains two or more paragraphs, which are all violated, then the *earliest* violation is the combined verdict. Contract disjunction is dual to contract conjunction, which means that it is always the *last* possible violation that is the combined verdict, but unlike contract conjunction, contract disjunction can introduce nondeterminism — even if both sub contracts are deterministic. In fact this is no surprise, since disjunction is inherently nondeterministic, however, in the restricted case where both sub contracts stipulate commitments on the same contract participant, disjunction will also be a deterministic operator — in this case disjunction corresponds to a *choice*.

The definition of the abstract contract model and the analysis of compositionality versus blame assignment provides for a better understanding of what makes contracts special. However, specifying contracts directly in the abstract model is not practical. Consequently, we need a specification language for writing contracts, which should support as many of the aspects mentioned earlier as possible, and which can be related to the abstract model via a mapping of specifications into abstract contracts. Furthermore, the language should be amenable to run-time monitoring. To this end, we propose the language *CSL* (Contract Specification Language), which overcomes the limitations of previous contract languages by supporting history sensitive contracts (i.e., contracts in which the exact content of commitments can depend on what has happened in the past) [1, 4], conditional commitments [2, 3, 4, 6, 13], contract templates [1], relative temporal constraints [1, 3, 4, 6], absolute temporal constraints [1, 2, 3, 4, 6], contrary-to-duty/reparation

clauses [2, 4, 13], and potentially infinite contracts [1, 13]. Furthermore, *CSL* contracts are by construction deterministic, which by virtue of the mapping into the semantic model means that contract violations can always be deterministically attributed to contract participants. For run-time monitoring of *CSL* contracts, we have defined a rewrite-based algorithm<sup>2</sup>, inspired by the algorithm of Andersen et al. [1]. The appeal of rewriting is that running contracts are conceptually no different from initial contracts, which entails that any analysis applicable to initial contracts are also applicable to running contracts.

## 2 Trace-based contract model

We strive to make our contract model as general as possible, and we therefore parametrize the model over *signatures*, instead of fixing the model to for instance a business ontology. A signature can be thought of as the *vocabulary* used in a contract, and formally it is a triple,  $S = (\mathcal{K}, \text{ar}, \mathcal{T})$ , where  $\mathcal{K}$  is a set of *action kinds*, with associated arities and types,  $\text{ar} : \mathcal{K} \rightarrow \mathcal{T}^*$ , and  $\mathcal{T}$  is the set of all types. An *action* has the form  $k(\vec{d})$ , where  $k \in \mathcal{K}$ ,  $\text{ar}(k) = (\tau_1, \dots, \tau_n)$ , and  $\vec{d} \in \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket$ . ( $\llbracket \tau \rrbracket$  denotes the domain of type  $\tau$ , e.g.,  $\llbracket \text{Int} \rrbracket = \mathbb{Z}$ .)  $\mathcal{A}$  denotes the set of all actions (over signature  $S$ ). Actions represent that *something happens*, but not who is responsible and when the action takes place. Consequently, we introduce *events*: an event is a triple,  $\langle t, a, k(\vec{d}) \rangle$ , where  $t \in \mathbb{T}$  is a *time stamp*,  $a \in \mathbb{A}$  is the *agent* responsible for the action, and  $k(\vec{d})$  is an action.  $\mathbb{A}$  denotes the (potentially infinite) set of all agents, and  $\mathbb{T} = \mathbb{N}$  denotes the time domain. Time stamps are interpreted as UNIX time, so we will write time stamps as dates, e.g., 2010-10-01, rather than integers. For a finite set of agents,  $A \subseteq_{\text{fin}} \mathbb{A}$ ,  $\mathbb{E}_A = \mathbb{T} \times A \times \mathcal{A}$  denotes the set of events performable by agents  $A$ . As an example, we can define a signature for modeling payments,  $S_p = (\{\text{payment}\}, \text{ar}, \{\text{€}, \text{Agent}\})$ , where  $\text{ar}(\text{payment}) = (\text{€}, \text{Agent})$ ,  $\llbracket \text{€} \rrbracket = \mathbb{N}$ , and  $\llbracket \text{Agent} \rrbracket = \mathbb{A}$ . Then the event  $\langle 2010-10-01, a_1, \text{payment}(500, a_2) \rangle$  represents the action that agent  $a_1$  pays €500 to agent  $a_2$  on 2010-10-01. Note that events are always associated with an agent responsible for the action, which means that external events such as *force majeure* will have to be modelled using an explicit *nature* agent.

As mentioned in the introduction, the outcome of a contract is determined by the events that take place. We therefore lift events to *event traces*,  $Tr_A = \mathbb{E}_A^*$ , where we require that all traces must have strictly increasing timestamps, in order to guarantee a unique ordering of events.  $\varepsilon \in Tr_A$  denotes the empty trace, and for  $\sigma \in Tr_A$  and  $t \in \mathbb{T}$ ,  $\sigma_t \in Tr_A$  denotes the trace that is the largest prefix of  $\sigma$  with all time stamps at most  $t$ . Intuitively then, a contract must partition traces into those that are conforming, and those that are nonconforming, which is the approach taken by for instance Andersen et al. [1] and Kyas et al. [5]. However, as argued in the introduction, it is crucial that all contract violations be attributed to a nonempty subset of the parties involved in the contract — we therefore generalize the traditional 0/1 outcome: for  $A \subseteq_{\text{fin}} \mathbb{A}$ , the set of *verdicts* is defined by  $\mathbb{V}^A = \{(t, B) \mid t \in \mathbb{T}, \emptyset \neq B \subseteq A\} \cup \{\top\}$ .  $\top$  denotes contract conformance, and contract violations are associated with a set of agents and the time of violation, denoted  $(t, B)$ . The reason why contract conformance is not associated with a point in time is that in some cases conformance only takes place “at infinity”:

**Paragraph 1.** Agent  $a$  is prohibited from performing action  $k(\vec{d})$ .

In the example above, any violation will take place at the time where  $a$  actually performs the forbidden action, but contract conformance is only guaranteed when we know that  $a$  never performed the action (i.e., at infinity). On the other hand, we require that all violations be associated with a *finite* point in time, which means that contract violations cannot take place at infinity. This restriction effectively means that

<sup>2</sup>Due to lack of space, we do not present the algorithm here. A copy of a technical report with all details can be obtained by contacting the author.

we do not allow contracts to define *liveness properties*, e.g.

**Paragraph 1.** Agent  $a$  must perform the action  $k(\vec{d})$  eventually.

We see this as a natural restriction, since the main purpose of formalizing contracts is to run-time monitor their execution, and hence it only makes sense if contract violations can be detected in finite time. We now have sufficient definitions to make the intuition of a contract formal: a contract involving agents  $A \subseteq_{\text{fin}} \mathbb{A}$  over the signature  $S$ , is a function,  $c : Tr_A \rightarrow \mathbb{V}^A$ , which must satisfy

$$\forall \sigma \in Tr_A. c(\sigma) = (t, B) \Rightarrow (\forall \sigma' \in Tr_A. \sigma_t = \sigma'_t \Rightarrow c(\sigma') = (t, B)). \quad (1)$$

(All actions in the traces,  $\sigma \in Tr_A$ , are required to be over the signature  $S$ .) The first thing we note about the definition is that contracts are *deterministic*, i.e.,  $c$  is a function and not in general a relation, and contracts do not model situations where more breaches occur at different time points, cf. the discussion in the introduction. The next thing we note about the definition of contracts is that event traces are considered *complete* (or maximal), which means that they represent a full history of what has happened. This means that for instance the empty trace,  $\varepsilon$ , represents that nothing has ever happened, and a trace,  $\sigma$ , where there are no events after time  $t$  means that nothing ever happened after that time point. Informally then, the condition (1) states that a contract violation at time  $t$  can only depend on what has (not) happened up until time  $t$ , i.e., it is a sanity check that the verdict of violation cannot depend on what happens in the future. Unlike the trace model of for instance LTL, traces are always *finite*, but it follows from the condition (1) that extending the model to include also infinite traces does not make the model more expressive (the verdict on an infinite trace will be uniquely determined by the verdicts on all finite prefixes).

Using the abstract definition of contracts and the payment signature,  $S_p$ , from earlier, it is now possible to define for instance a payment obligation as follows:

$$c(\sigma) = \begin{cases} \top, & \text{if } \langle t, \text{Buyer}, \text{payment}(500, \text{Seller}) \rangle \in \sigma, \text{ with } t \leq 2010-10-01 \\ (2010-10-01, \{\text{Buyer}\}), & \text{otherwise} \end{cases}.$$

$c$  represents the obligation that Buyer has to pay €500 to Seller on or before 2010-10-01. As argued in the introduction, contract conjunction is inherent in nearly all contracts, albeit often implicitly:

**Paragraph 1.**  $a_1$  must pay  $a_2$  €500 on or before 2010-10-01.

**Paragraph 2.**  $a_2$  must deliver to  $a_1$  1 iPad on or before 2010-10-15.

Assuming we have constructed sub verdicts for the two paragraphs,  $v_1$  and  $v_2$ , it is clear when the combined contract has been fulfilled: exactly when the two sub contracts are fulfilled (i.e.,  $v_1 = v_2 = \top$ ). However, in case one (or both) sub contracts are not fulfilled, we argued that the *earlier* violation should be the combined verdict, and we therefore define the conjunction of two verdicts as follows:

$$v_1 \otimes v_2 = \begin{cases} \top, & \text{if } v_1 = v_2 = \top \\ (t_1, B_1), & \text{if } v_1 = (t_1, B_1) \wedge (v_2 = \top \vee (v_2 = (t_2, B_2) \wedge t_1 < t_2)) \\ (t_2, B_2), & \text{if } v_2 = (t_2, B_2) \wedge (v_1 = \top \vee (v_1 = (t_1, B_1) \wedge t_2 < t_1)) \\ (t, B_1 \cup B_2), & \text{if } v_1 = (t, B_1) \wedge v_2 = (t, B_2) \end{cases}.$$

It then remains to define how to construct the sub verdicts,  $v_1$  and  $v_2$ , for sub contracts  $c_1$  and  $c_2$ , given a trace  $\sigma$ . The immediate solution is  $v_1 = c_1(\sigma)$  and  $v_2 = c_2(\sigma)$ , but the problem with this definition is that events are treated *nonlinearly*, so for instance the contract

**Paragraph 1.**  $a_1$  must pay  $a_2$  €100 on or before 2010-10-01.

**Paragraph 2.**  $a_1$  must pay  $a_2$  €100 on or before 2010-10-15.

will be fulfilled by the trace  $\sigma = \langle 2010-10-01, a_1, \text{payment}(100, a_2) \rangle$ , even though the intention is that  $a_1$  must pay €100 twice. We therefore need a way of partitioning the trace  $\sigma$  into two parts: those events that belong to  $c_1$  and those that belong to  $c_2$ . So the question is how to perform such splitting, and the approach we take is simple: the agents generating the events should decide. In the example above, if  $a_1$  pays €100 to  $a_2$  on 2010-10-01, then *probably*  $a_1$  would like the early payment to be discharged, but we really cannot know. By making  $a_1$  in charge of the explicit choice, we avoid having to make potentially unintended choices in the semantics of contract conjunction, and in fact this definition also matches what often happens in real world contracts, where identical payments are made distinguishable by assigning a unique reference number to each payment. Consequently, we extend events with a *choice* parameter (we omit the details here), and define contract conjunction as  $(c_1 \otimes c_2)(\sigma) = c_1(\sigma_1) \otimes c_2(\sigma_2)$ , where  $c_1$  and  $c_2$  are contracts involving agents  $A$  over the signature  $S$ , and  $\sigma_1$  and  $\sigma_2$  partition the original trace  $\sigma$  with respect to the choices made.

For contract disjunction, we proceed as in the case of contract conjunction, and assume we have constructed sub verdicts for the two sub contracts,  $v_1$  and  $v_2$ . As was the case for contract conjunction, it is easy to define conformance: a disjunction is fulfilled exactly when at least one of the two sub contracts are fulfilled (i.e.,  $v_1 = \top$  or  $v_2 = \top$ ). Dually to contract conjunction, if both sub contracts are violated, then the combined verdict should be the *last* violation, but if both violations take place at the same point in time by different agents, then we cannot combine the verdicts to a deterministic verdict. This makes disjunction an inherently nondeterministic operator, but since the model is restricted to deterministic contracts, verdict disjunction becomes a partial operator:

$$v_1 \oplus v_2 = \begin{cases} v_1, & \text{if } v_2 = \top \vee v_1 = v_2 \\ v_2, & \text{if } v_1 = \top \vee v_1 = v_2 \\ (t_1, B_1), & \text{if } v_1 = (t_1, B_1) \wedge v_2 = (t_2, B_2) \wedge t_1 > t_2 \\ (t_2, B_2), & \text{if } v_1 = (t_1, B_1) \wedge v_2 = (t_2, B_2) \wedge t_1 < t_2 \\ \text{undefined,} & \text{otherwise} \end{cases} .$$

The definition matches the intuition explained above, and only in the case where the two verdicts are violations at the same point in time by different agents, is the disjunction undefined. Contract disjunction can now be defined as a partial operator,  $(c_1 \oplus c_2)(\sigma) = c_1(\sigma) \oplus c_2(\sigma)$ , where  $c_1$  and  $c_2$  are contracts involving agents  $A$  over the signature  $S$ , and  $c_1 \oplus c_2$  is defined when  $c_1(\sigma) \oplus c_2(\sigma)$  is defined, for all  $\sigma \in Tr_A$ . In the restricted case where the disjunction is a choice by one agent, it becomes well-defined, because in that case only one agent can be blamed — which can be utilized to encode contrary-to-duty:

**Paragraph 1.**  $a_1$  must pay  $a_2$  €100 on or before 2010-10-01.

**Paragraph 2.** If  $a_1$  violates Paragraph 1,  $a_1$  must pay  $a_2$  €110 on or before 2010-10-15.

In the example above, if  $a_1$  does not pay on the early nor the late deadline, then the combined verdict,  $(2010-10-01, \{a_1\}) \oplus (2010-10-15, \{a_1\})$ , is a violation at the time of the late deadline.

We argued in the introduction that run-time monitoring of contracts is crucial. Consequently, we now define (abstractly) what a run-time monitor is, using a many-valued semantics as is conventional in run-time monitoring [7]. In the abstract contract model, traces are assumed to be complete, in the sense that the finite traces describe all that has ever happened. For run-time monitoring however, traces are necessarily never complete, as events may be generated continuously. Therefore, even though both the abstract trace model and the run-time monitor deal with finite traces, we consider them to be semantically different. A run-time monitor for a contract  $c$  involving agents  $A$  over the signature  $S$ , is defined as a

function,  $\text{RM}_c : \text{Tr}_A \times \mathbb{T} \rightarrow \mathbb{V}^A \cup \{\text{Nullable}, ?\}$ , which must satisfy

$$\text{RM}_c(\sigma, t) = \begin{cases} \top, & \text{if } \forall \sigma' \in \text{Tr}_A. \sigma'_t = \sigma \Rightarrow c(\sigma') = \top \\ \text{Nullable}, & \text{if } \exists \sigma' \in \text{Tr}_A. \sigma'_t = \sigma \wedge c(\sigma') \neq \top \wedge c(\sigma) = \top \\ (t', B), & \text{if } c(\sigma) = (t', B) \wedge t' \leq t \\ ?, & \text{otherwise} \end{cases}.$$

The verdict of a run-time monitor should be interpreted as follows: (i) if  $\text{RM}_c(\sigma, t) = \top$ , then nobody will violate the contract, no matter what happens in the future; (ii) if  $\text{RM}_c(\sigma, t) = \text{Nullable}$ , then nobody will violate the contract, provided that nothing happens in the future; (iii) if  $\text{RM}_c(\sigma, t) = (t', B)$  then the contract has been violated by agents  $B$  at time  $t' \leq t$ ; and (iv) if  $\text{RM}_c(\sigma, t) = ?$  then we cannot yet say who (if any) have violated the contract. Most interestingly, run-time monitoring is not defined on partial traces only, but instead on partial traces and a *current point in time*, which is therefore required to be at least the time of the last event in the partial trace. This is because contracts may be violated due to some deadline not being met, in which case a violation should be reported. If we consider the contract  $c$  from page 4, then a violation should be reported if nothing happens until 2010-10-01, since in that case Buyer has failed to pay Seller; and indeed by definition  $\text{RM}_c(\varepsilon, 2010-10-01) = \{(2010-10-01, \{\text{Buyer}\})\}$ . But had we instead defined  $\text{RM}_c$  only on partial traces, then the run-time monitor could only report a violation once some (random) event took place after 2010-10-01 — and such event might never happen. It is worth mentioning that the abstract definition of run-time monitoring satisfies the *impartiality* maxim [7], i.e., for all  $\sigma, \sigma' \in \text{Tr}_A$  with  $\sigma = \sigma'_t$ , we have that  $\text{RM}_c(\sigma, t) = \top \Rightarrow \text{RM}_c(\sigma', t') = \top$ , for all  $t' \in \mathbb{T}$ , and  $\text{RM}_c(\sigma, t) = (t'', B) \Rightarrow \text{RM}_c(\sigma', t') = (t'', B)$ , for all  $t' \in \mathbb{T}$ . However, the *anticipation* maxim [7] is not satisfied, since inevitable future violations are not reported until the future has been reached.

### 3 A contract specification language

We now present the contract specification language *CSL*, which by construction can be mapped into the abstract, trace-based model presented in the previous section (due to lack of space we do not present the semantics here). *CSL* is inspired by the language of Andersen et al. [1], but there are some major differences, the most important of which are: (i) *CSL* uses the trace model from the previous section, hence all contracts encoded in *CSL* have deterministic blame assignment; (ii) *CSL* is parametrized over signatures, so is not fixed to a particular contract domain; (iii) *CSL* permits internal conditionals, atomic obligations, and atomic permissions; (iv) *CSL* prohibits nondeterministic choice, general sequencing, and explicit failure; (v) *CSL* statically infers the agents involved in a contract via type checking; and (vi) *CSL* avoids potential deadlines in the past. The full grammar for *CSL* is presented below:

$p$	::=	letrec $\{f_i(\vec{x}_i) \langle \vec{y}_i \rangle = c_i\}_{i=1}^n$ in $c$ starting $t$	( <i>CSL</i> contract)
$c$	::=	empty	(No obligations nor rights)
		$ac$	(Atomic clause)
		$c_1$ and $c_2$	(Conjunction)
		$c_1$ or $c_2$	(Disjunction)
		if $e$ then $c_1$ else $c_2$	(Conditional)
		$f(\vec{e}) \langle \vec{e}_a \rangle$	(Template instantiation)
$ac$	::=	$\langle e_a \rangle k(\vec{x}) @ x$ where $e$ due $e_d$ then $c$	(Atomic obligation)
		if $\langle \vec{e}_a \rangle k(\vec{x}) @ x$ where $e$ due $e_d$ then $c_1$ else $c_2$	(Atomic permission)
$e$	::=	$x   d   eD   eW   e_1 \circ e_2   e_1 \prec e_2$	(Expressions, $\circ \in \{+, -, *, /, \text{andalso}\}$ , $\prec \in \{<, \leq, =\}$ )
$e_a$	::=	$x   a$	(Agent expressions)
$e_d$	::=	within $e_1$ after $e_2$	(Deadline expressions)

A *CSL* contract,  $p$ , consists of a set of mutually recursive template definitions, which can be referenced in the body of the contract. The body of a *CSL* contract constitutes a *clause*,  $c$ , together with an absolute point in time, which defines the starting point of the contract. The parameters of a template are divided into values,  $\vec{x}$ , and agents,  $\vec{y}$ . Value parameters are *dynamic*, meaning that they can be instantiated with actual values from earlier events, whereas agent parameters are *static*, meaning that all contract agents are defined in the top-level body of the *CSL* contract, and the participants do not change over time. Mutual recursion makes it possible to define potentially infinite contract in *CSL*. Clauses can be thought of as *pre contracts*, in the sense that they describe the normative content of a contract, but all deadlines in clauses are defined using relative measures such as “4 days”. The relative deadlines are lifted to absolute deadlines when the clause is used in a *CSL* contract context, thereby lifting the pre contract to a contract. *CSL* does not include deontic modalities, but instead uses a closed-world assumption: all actions that are obliged are automatically permitted, and all actions that are not permitted are automatically prohibited.

empty represents a trivially fulfilled clause, in which nothing is obliged nor permitted, and hence all actions are prohibited. Atomic clauses are divided into atomic *obligations* and atomic *permissions*. Instantiated atomic obligations have the form,  $\langle a \rangle k(\vec{x}) @ x$  where  $e$  due  $e_d$  then  $c$ , which should be read: “ $a$  must perform an action of kind  $k$ , which satisfies the condition  $e$  and the deadline  $e_d$ . After successful completion,  $c$  is the residual clause”. As mentioned above, deadlines are always relative — the reason we do not allow absolute deadlines such as 1st of October 2010 directly in clauses, is to avoid deadlines in the past, e.g., “first  $a_1$  must pay on 2010-10-01, and then  $a_2$  must deliver on 2010-09-01”. The vector of variables,  $\vec{x}$ , is bound to the actual  $k$ -values in the event, and  $x$  is bound to the relative time elapsed until the event is performed. The scope of  $\vec{x}$  is  $e$  and  $c$ , while the scope of  $x$  is  $c$  only. All deadlines in the continuation,  $c$ , are relative to the time of the  $k$ -event. Instantiated atomic permissions have the form if  $\langle \vec{a} \rangle k(\vec{x}) @ x$  where  $e$  due  $e_d$  then  $c_1$  else  $c_2$ , which should be read: “any agent  $a \in \vec{a}$  may perform an action of kind  $k$ , which satisfies the condition  $e$  and the deadline  $e_d$ , after which  $c_1$  is the residual clause. If the action is not performed within the deadline,  $c_2$  is the residual clause”. Contrary to atomic obligations, atomic permissions can in general include a nonempty set of agents, which is possible because an atomic permission cannot be violated by not performing it (in that case  $c_2$  dictates the further commitments).  $c_1$  and  $c_2$  and  $c_1$  or  $c_2$  denote clause conjunction and disjunction, respectively, as described in Section 2. The type system of *CSL*, which we also do not present here, ensures that  $c_1$  and  $c_2$  only stipulate commitments on the same agent in clause disjunctions, which is sufficient to guarantee that the semantic disjunction,  $\oplus$ , is always defined. if  $e$  then  $c_1$  else  $c_2$  represents (internal) clause branching, where the branching condition  $e$  can be computed directly without having to wait for external input (i.e., events), and finally  $f(\vec{e})\langle \vec{e}_a \rangle$  denotes instantiation of template  $f$ , where  $\vec{e}$  are clause parameters, and  $\vec{e}_a$  are agent parameters. We conclude with an example of a contract encoded in *CSL* (we omit the *freeze variables* in atomic clauses, since they are not used, and use implicit empty continuations):

```

letrec
sale(deliveryDeadline, goods, amount)<buyer, seller> =
  // First seller must deliver goods
  <seller> TransferAndDeliver(g,r) where g == goods andalso r == buyer due within deliveryDeadline after OD then
  // Buyer must pay half upon receipt
  <buyer> Payment(a,r) where a == amount / 2 andalso r == seller due within OD after OD then
  ((// Remainder due within 30 days of receipt
    <buyer> Payment(a,r) where a == amount / 2 andalso r == seller due within 30D after OD
    or
    // Fine in case of late payment
    <buyer> Payment(a,r) where a == 1.1 * (amount / 2) andalso r == seller due within 14D after 30D)
  and
  // Buyer has 14 days to claim for damages
  if <buyer> ClaimForDamages(g,r) where g == goods andalso r == seller due within 14D after OD then
  // In case the goods are damaged, Seller must repay Buyer
  <seller> Payment(a,r) where a == amount andalso r == buyer due within 7D)
in
sale(OD, "iPad", 500)<Tom, Apple> starting 2010-10-01

```

The encoding assumes a signature with `TransferAndDeliver`, `Payment`, `ClaimForDamages`  $\in \mathcal{K}$ , and the rather small example illustrates conditional commitments (Seller's obligation to repay Buyer in case of damaged goods), relative and absolute temporal constraints (immediate payment after delivery and delivery at a specific date, respectively), contrary-to-duty obligations (fine in case of late payment), and arithmetic expressions (calculation of fine).

**Acknowledgements.** The author wishes to thank Dr. Felix Klaedtke and Dr. Eugen Zalinescu for numerous and enlightening discussions, and the anonymous referees for valuable feedback. In addition, the author expresses his gratitude to Prof. Dr. David Basin for the invitation to visit ETH Zürich.

## References

- [1] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen & Christian Stefansen (2006): *Compositional specification of commercial contracts*. *International Journal on Software Tools for Technology Transfer (STTT)* 8(6), pp. 485–516.
- [2] Abdel Boulmakoul & Mathias Sall (2002): *Integrated Contract Management*. Technical Report, HP Laboratories Bristol.
- [3] Andrew Goodchild, Charles Herring & Zoran Milosevic (2000): *Business Contracts for B2B*. In: *Proceedings of the CAISE '00 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing (ISDO 2000)*, pp. 63–74.
- [4] Guido Governatori & Zoran Milosevic (2006): *A Formal Analysis of a Business Contract Language*. *Int. J. Cooperative Inf. Syst.* 15(4), pp. 659–685.
- [5] Marcel Kyas, Cristian Prisacariu & Gerardo Schneider (2008): *Run-Time Monitoring of Electronic Contracts*. In: *ATVA '08: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, Springer-Verlag, Berlin, Heidelberg, pp. 397–407.
- [6] Ronald M. Lee (1988): *A Logic Model for Electronic Contracting*. *Decision Support Systems* 4(1), pp. 27–44.
- [7] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *Journal of Logic and Algebraic Programming* 78(5), pp. 293–303. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).
- [8] Carlos Molina-jimenez, Carlos Molina-jimenez, Santosh Shrivastava, Santosh Shrivastava, Ellis Solaiman, Ellis Solaiman, John Warne & John Warne (2004): *Run-time Monitoring and Enforcement of Electronic Contracts*. *Electronic Commerce Research and Applications* 3, p. 2004.
- [9] Nir Oren, Sofia Panagiotidi, Javier Vázquez-Salceda, Sanjay Modgil, Michael Luck & Simon Miles (2009): *Towards a Formalisation of Electronic Contracting Environments*. In: *Coordination, Organizations, Institutions and Norms in Agent Systems IV*, Springer-Verlag, Berlin, Heidelberg, pp. 156–171.
- [10] Gordon J. Pace & Gerardo Schneider (2009): *Challenges in the Specification of Full Contracts*. In: *IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods*, Springer-Verlag, Berlin, Heidelberg, pp. 292–306.
- [11] Vishal Patel & Christopher J. Dwyer (2007): *Contract Lifecycle Management and the CFO: Optimizing Revenues and Capturing Savings*. Technical Report, Aberdeen Group.
- [12] Simon Peyton-Jones & Jean-Marc Eber (2003): *How to write a financial contract*. In: Jeremy Gibbons & Oege de Moor, editors: *The Fun of Programming*, chapter 6, Cambridge University Press, New York, NY, USA, pp. 105–130.
- [13] Cristian Prisacariu & Gerardo Schneider (2007): *A Formal Language for Electronic Contracts*. In: *Formal Methods for Open Object-Based Distributed Systems*, Springer-Verlag, Berlin, Heidelberg, pp. 174–189.
- [14] Lai Xu (2004): *A multi-party contract model*. *SIGecom Exch.* 5(1), pp. 13–23.