The Journal of Logic and Algebraic Programming 81 (2011) 72–98

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

A trace-based model for multiparty contracts

Tom Hvitved [a,*], Felix Klaedtke [b], Eugen Zălinescu [b]

[a] Department of Computer Science, University of Copenhagen, Denmark
[b] Computer Science Department, ETH Zurich, Switzerland

ARTICLE INFO

Article history:
Available online 22 August 2011

Keywords:
Contracts
Blame assignment
Contract specification language
Operational semantics
Run-time monitoring

ABSTRACT

In this article we present a model for multiparty contracts in which contract conformance is defined abstractly as a property on traces. A key feature of our model is blame assignment, which means that for a given contract, every breach is attributed to a set of parties. We show that blame assignment is compositional by defining contract conjunction and contract disjunction. Moreover, to specify real-world contracts, we introduce the contract specification language CSL with an operational semantics. We show that each CSL contract has a counterpart in our trace-based model and from the operational semantics we derive a run-time monitor. CSL overcomes limitations of previously proposed formalisms for specifying contracts by supporting: (history sensitive and conditional) commitments, parametrised contract templates, relative and absolute temporal constraints, potentially infinite contracts, and in-place arithmetic expressions. Finally, we illustrate the general applicability of CSL by formalising in CSL various contracts from different domains.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

Contracts are legally binding agreements between parties and in e-business it is particularly crucial to automatically check conformance to them, for example for minimising financial penalties. The Aberdeen Group [20,21] has recently identified *contract lifecycle management* (CLM) as a key methodology in e-business: CLM is a broad term used to cover the activities of systematically and efficiently managing contract creation, contract negotiation, contract approval, contract analysis, and contract execution. Monitoring the execution of contracts constitutes the primary incentive for enterprises to use CLM, since it enables qualified decision making and makes it possible to issue reminders for upcoming deadlines, which may lead to a significant decrease of financial loss due to noncompliance:

"[...] the average savings of transactions that are compliant with contracts is 22%." [20, page 1]

Consequently, several systems that implement the CLM methodology have been deployed.[1] More traditional *enterprise resource planning* (ERP) systems such as Microsoft Dynamics NAV [16] and Microsoft Dynamics AX [15] are also used for

[*] Corresponding author.
E-mail address: hvitved@diku.dk (T. Hvitved)

[1] Examples of such systems include (all URLs retrieved on May 18th 2011):
- Blueridge Software *Contract Assistant*, http://www.blueridgesoftware.bz.
- CobbleStone Systems *ContractInsight*, http://www.cobblestonesystems.com.
- Moai *CompleteSource Contract Management*, http://www.moai.com.
- Ecteon *Contraxx*, http://www.ecteon.com.
- Emptoris *Contract Management Solutions*, http://www.emptoris.com.
- Great Minds Software *Contract Advantage*, http://www.greatminds-software.com.
- IntelliSoft Group *IntelliContract*, http://www.intellisoftgroup.com.
- Ketera *Contract Management*, http://www.ketera.com.
- Open Text *Contract Management*, http://www.opentext.com.

1567-8326/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2011.04.010

Paragraph 1. Seller agrees to transfer and deliver to Buyer, on or before 2011-01-01, the goods: 1 laser printer.

Paragraph 2. Buyer agrees to accept the goods and to pay a total of €200 for them according to the terms further set out below.

Paragraph 3. Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery.

Paragraph 4. If Buyer fails to pay the second half within 30 days, an additional fine of 10% has to be paid within 14 days.

Paragraph 5. Upon receipt, Buyer has 14 days to return the goods to Seller in original, unopened packaging. Within 7 days thereafter, Seller has to repay the total amount to Buyer.

Fig. 1. A sales contract between a buyer and a seller.

managing business agreements. However, a shortcoming of existing CLM and ERP systems is that contracts are dealt with in an ad hoc manner rather than as first-class objects. In fact, the before mentioned studies by the Aberdeen Group [20,21] suggest the use of a domain-specific language as the basis for automated CLM. Although various authors have proposed domain-specific languages for representing contracts [2,3,7,9,12,22,26], constructing a widely applicable contract specification language remains a challenge [19]. One reason is that contracts involve many different aspects like absolute temporal constraints (as in deadlines), relative temporal constraints (for imposing an ordering on the occurrence of certain actions), reparation clauses, conditional commitments, different deontic modalities [28] (such as obligations and permissions), and repetitive patterns. In order to make some of these aspects concrete, consider the contract in Fig. 1, which we will use as a running example in the remainder of this article. This sample contract involves both obligations (Paragraph 1), permissions (Paragraph 5), absolute deadlines (Paragraph 1), relative deadlines (Paragraph 3), and reparation activities (Paragraph 4). Additionally, it involves data dependencies between paragraphs, for example the payment amount in Paragraph 4 depends on the amount defined in Paragraph 3.

Besides being able to capture the various aspects found in contracts mentioned above, a contract specification language should also be amenable to automatic analysis. In particular, the language should support *run-time monitoring* [13] of contracts, that is reporting of (potential) contract breaches during execution—for instance as the result of passing a deadline or performing a forbidden action. Furthermore, in case of noncompliance the run-time monitor should be able to assign *blame* to one or more of the parties involved in the contract, rather than simply reporting noncompliance without specifying who is responsible for the breach of contract. Surprisingly, even though run-time monitoring of contracts has been studied extensively [2,7,9,17,26,31], blame assignment has not been given much attention yet. To the best of our knowledge only Xu [31] investigates blame assignment though not from the viewpoint of run-time monitoring, but rather from an off-line viewpoint where blame has to be determined from a set of unfulfilled, dependent commitments.

In this article, we present a contract specification language that targets at naturally formalising and monitoring contracts. In particular, contracts formalised in our language can directly be monitored, and in case of noncompliance the monitor assigns blame to the responsible contract parties. Although our focus is on business contracts, our language is not essentially restricted to this particular application area.

1.1. Breach of contract and blame assignment

A first question that arises when designing such a contract specification language is what constitutes a breach of contract? Returning to the example contract in Fig. 1, one can think of several scenarios which arguably constitute breaches of contract:

(1) Seller fails to deliver to Buyer on time.
(2) Seller delivers on time, Buyer pays first half on delivery, but Buyer does not pay second half on time.
(3) Seller delivers on time, Buyer pays first half on delivery, Buyer does not pay second half on time, and Buyer does not pay the additional fine on time.

Clearly, the first scenario represents a breach of contract, and Seller is to be blamed for not delivering the goods to Buyer. In the second scenario, it is less clear, since Buyer has violated Paragraph 3, but depending on whether the extended deadline has passed, Buyer may or may not have breached the contract. Finally, in the last scenario it is clear that Buyer has breached the contract, but it is perhaps less clear whether violating Paragraph 3 or Paragraph 4 (or both) constitutes the breach of contract.

The approach we take is that of *fundamental breaches*: a breach of contract takes place only when a violation happens, from which the contract cannot recover, and from which it therefore does not make sense to continue executing the contract. In terms of run-time monitoring, a breach of contract hence takes place only when it is impossible to complete a conforming

- 8over8 *ProCon Contract Management*, http://www.8over8.com.
- SAP *SAP CLM*, http://www.sap.com.
- Procuri *TotalContracts*, http://www.procuri.com.
- Upside Software *UpsideContract*, http://www.upsidesoft.com.

execution. With this rather informal definition of contract breach, we see that the first scenario constitutes indeed a breach of contract. Regarding the second scenario, it depends whether Buyer will pay the fine or not, as only neglecting to pay the fine constitutes a breach of contract. Thus scenario (2) does not yet represent a breach, in contrast to the last scenario (3).

We deliberately use the term *breach* rather than *violation* in order to distinguish our concept of (fundamental) breach from the more traditional notion of violation known from standard deontic logic (SDL) with contrary-to-duty obligations [25]. In the context of SDL, it is tempting to encode reparation clauses like the one in Paragraph 4 in the form of a contrary-to-duty obligation. Yet, with such an encoding there is an implicit agreement that the *primary* obligation (Paragraph 3) should be complied with first and foremost, and only complying with the reparation obligation constitutes a violation, even though—from a contractual point of view—the contract is fulfilled.

A classical example which illustrates the subtle, but important, difference is the "gentle murderer": do not kill, but if you kill, kill gently [5]. The gentle murderer is an actual contrary-to-duty obligation, because there is an implicit agreement that you should not kill—only if you have no other options than killing, then at least you should do so gently.

We argue, however, that contracts should not contain implicit agreements, in particular because parties may have conflicting interests. Hence if one party wishes to impose that an obligation be primary, then the only way to do so is by making sure that there is an incentive for the responsible (counter) party to perform the primary obligation, for example by imposing a penalty for complying only with the reparation obligation. Hence the gentle murderer, as a contract, would be: do not kill, but if you kill, kill gently and go to jail. Attaching penalties to violations yields new obligations. Violating such an obligation might result in new obligations until either all obligations are fulfilled or eventually a breach of contract is reached. For the example, killing non-gently represents a breach of contract. Killing gently and not going to jail also represents a breach of contract. However, killing gently and going to jail is not a breach of contract. Note that the consequences of breaching the contract are not specified.

Ideally, blame assignment should be *deterministic*, that is it should uniquely determine the parties responsible for a breach. However, not all contracts allow for deterministic blame assignment, as illustrated by the following scenario: if one paragraph specifies that Alice has to fulfil an obligation by time τ, and another paragraph that Bob has to fulfil another obligation by the same time τ, and the contract only asks for conformance with one of the paragraphs, then we are in a delicate situation—who is to blame if neither Alice nor Bob has fulfilled her/his obligation?[2] Contracts involving disjunction, such as this one, lead to nondeterministic blame assignment. In other words, such contracts are ambiguous. For simplicity, we choose not to model them, except in the special cases when the same parties are blamed in both subcontracts. Our choice is also motivated by the fact that such scenarios rarely correspond to real-world contracts.

1.2. Contributions and organisation

We see our main contributions as follows. First, we present an abstract, trace-based model for contracts that has blame assignment at its core. Furthermore, our model supports modular composition of contracts by contract conjunction and disjunction. Second, we introduce the contract specification language (CSL) that fits naturally—by means of a mapping—to our abstract model, and that overcomes many of the limitations of previous specification languages for contracts. Third, we describe a run-time monitoring algorithm for CSL specifications obtained as a by-product of the reduction semantics of CSL.

The remainder of this article is structured as follows. In Section 2 we present our abstract, trace-based model for contracts, relying on the informal notion of contract breach and blame assignment described above. We show how our model encodes various high-level aspects, such as obligations, permissions, and reparation clauses without relying on such notions. We also provide operators for composing contracts and show that they fulfil desirable algebraic properties. In Section 3 we introduce the contract specification language CSL, together with a formal semantics which maps CSL into our abstract, trace-based contract model. Furthermore, from the small-step, reduction-based semantics of CSL, we derive a run-time monitoring algorithm. We also demonstrate the applicability of CSL by means of several example contracts. We discuss related work in Section 4 and we draw conclusions in Section 5. The appendix contains additional proof details.

2. Trace-based contract model

Trace-based contract models have been proposed before [2,11], but unlike our model, those models partition traces into conforming and nonconforming traces, without taking blame assignment into account. A trace is a sequence of actions that represent the *complete* history of actions that have occurred during the execution of a contract. In order to capture real-time aspects, and not only relative temporality, actions of a trace are timestamped. In this article we ignore how actions are generated, and neither do we model how parties agree that actions have taken place—the latter would usually involve a hand-shaking protocol, which is outside the scope of our work. For the purpose of defining contracts, we hence assume a trace of timestamped actions is given.

[2] We leave it to the reader to ponder whether blaming neither of the two, or blaming both of them is acceptable. Our view is that neither option is acceptable.

2.1. Notation and terminology

Before presenting our contract model, we fix the notation and terminology that we use in the remainder of the text. Throughout this article, P denotes the set of *parties*, A the set of *actions*, and Ts the set of *timestamps*. The sets P and A can be finite or infinite but we require that they are both non-empty. We require that Ts is totally ordered by the relation $\leq$, and that Ts has a least element and that no element in the set is an upper bound, that is for all $\tau \in \mathsf{Ts}$ there is some $\tau' \in \mathsf{Ts}$ such that $\tau \neq \tau'$ and $\tau \leq \tau'$. In the following, for representation issues, we assume that $\mathsf{Ts} = \mathbb{N}$.

We write a finite sequence σ over an alphabet Σ as $\langle \sigma[0], \sigma[1], \ldots, \sigma[n-1] \rangle$, where $\sigma[i] \in \Sigma$ denotes the $(i+1)$st letter of σ. Its length is n and denoted by $|\sigma|$. In particular, $\langle\rangle$ denotes the empty sequence which has length 0. Analogously, an infinite sequence σ over Σ is written as $\langle \sigma[0], \sigma[1], \sigma[2], \ldots \rangle$ with $\sigma[i] \in \Sigma$, for every $i \in \mathbb{N}$. The length of an infinite sequence σ is $|\sigma| = \infty$. We write $\sigma \sqsubset \sigma'$ if the sequence σ is a finite prefix of the sequence σ', that is if σ is finite and there is a sequence σ'' such that $\sigma' = \sigma\sigma''$, where $\sigma\sigma''$ denotes the concatenation of the sequences σ and σ''.

An *event* is a tuple (τ, α), where $\tau \in \mathsf{Ts}$ is a timestamp and $\alpha \in \mathsf{A}$ an action. We write $\mathrm{ts}(\epsilon)$ for the timestamp of an event $\epsilon = (\tau, \alpha)$, that is $\mathrm{ts}(\epsilon) = \tau$. A *trace* σ is a finite or infinite sequence of events where the sequence of timestamps are

(1) increasing, that is $\mathrm{ts}(\sigma[i]) \leq \mathrm{ts}(\sigma[j])$ for all $i, j \in \mathbb{N}$ with $i \leq j < |\sigma|$, and
(2) progressing for infinite traces, that is for all $\tau \in \mathsf{Ts}$ there is some $i \in \mathbb{N}$ such that $\mathrm{ts}(\sigma[i]) \geq \tau$ whenever $|\sigma| = \infty$.

We denote the set of all traces by Tr, and the subset of finite traces by $\mathsf{Tr}_{\mathsf{fin}}$, that is $\mathsf{Tr}_{\mathsf{fin}} = \{\sigma \in \mathsf{Tr} \mid |\sigma| \neq \infty\}$. Tr^{τ} denotes the subset of traces where all timestamps are at least τ, and similarly for $\mathsf{Tr}^{\tau}_{\mathsf{fin}}$. For a finite non-empty trace σ, the timestamp of the last event in σ is denoted by $\mathrm{end}(\sigma)$, and for the empty trace, we define $\mathrm{end}(\langle\rangle) = 0$.

For a trace $\sigma \in \mathsf{Tr}$ and a timestamp $\tau \in \mathsf{Ts}$, σ_τ denotes the longest prefix of σ with $\mathrm{end}(\sigma_\tau) \leq \tau$. This prefix exists, since the properties (1) and (2) ensure that there are only finitely many prefixes $\sigma' \sqsubset \sigma$ with $\mathrm{end}(\sigma') \leq \tau$.

Finally, we denote the domain of a (partial) function f by $\mathrm{dom}(f)$, that is $\mathrm{dom}(f)$ is the set of elements a for which $f(a)$ is defined. For a function f and a set $X \subseteq \mathrm{dom}(f)$, $f|_X$ denotes the restriction of f to X.

2.2. Contracts

We capture blame assignment by generalising the outcome of a contract execution from a binary result (conformance or nonconformance) to *verdicts*, defined as elements of the set

$$\mathsf{V} = \{\checkmark\} \cup \{(\tau, B) \mid \tau \in \mathsf{Ts} \text{ and } B \text{ is a non-empty finite subset of } \mathsf{P}\},$$

where $\checkmark$ represents *contract conformance*, that is no one is to be blamed, and (τ, B) represents a *breach of contract* at time τ by the parties in B. Whenever $|B| > 1$ then multiple parties have breached the contract simultaneously. For instance, both parties of a barter deal may breach the contract if neither hands over the agreed goods.

A contract is defined as a function that maps traces to verdicts:

Definition 1. Let P be a non-empty and finite subset of P. A *contract* between parties P, starting at time $\tau_0 \in \mathsf{Ts}$, is a function $\mathsf{c} : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ that satisfies the following conditions for all $\sigma \in \mathsf{Tr}^{\tau_0}$ and $(\tau, B) \in \mathsf{V}$:

$$\text{if } \mathsf{c}(\sigma) = (\tau, B) \text{ then } B \subseteq P \text{ and } \tau \geq \tau_0, \tag{1}$$

and

$$\text{if } \mathsf{c}(\sigma) = (\tau, B) \text{ then } \mathsf{c}(\sigma') = (\tau, B), \text{ for all } \sigma' \in \mathsf{Tr}^{\tau_0} \text{ with } \sigma_\tau = \sigma'_\tau. \tag{2}$$

The contract for which all traces are conforming is denoted $\mathsf{c}_\checkmark$, that is $\mathsf{c}_\checkmark$ is the function with $\mathsf{c}_\checkmark(\sigma) = \checkmark$, for all $\sigma \in \mathsf{Tr}^{\tau_0}$.

The definition entails that contracts are deterministic, as c is a function. Since traces are considered complete, condition (2) guarantees that a breach at time τ only depends on what has (and has not) happened up until time τ. Moreover, the verdict of a contract can only depend on what has happened after the contract started.

Example 1. We illustrate our contract model by representing Paragraph 1 in Fig. 1 as a contract $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$, for a suitable τ_0. As the paragraph only defines an obligation on the party Seller, we define c_1 as a contract "between" {Seller} with

$$\mathsf{c}_1(\sigma) = \begin{cases} \checkmark & \text{if } \sigma[i] = (\tau, \text{delivery}), \text{ for some } i \in \mathbb{N} \text{ and } \tau \in \mathsf{Ts} \text{ with } i < |\sigma| \text{ and } \tau \leq \tau_d, \\ (\tau_d, \{\text{Seller}\}) & \text{otherwise.} \end{cases}$$

The action delivery represents the delivery of goods to the party Buyer and τ_d represents the deadline 2011-01-01. Note that dates like 2011-01-01 can be easily interpreted as non-negative integers by taking for instance the corresponding UNIX time. It is easy to check that c_1 satisfies the properties of Definition 1.

2.3. Contract conformance on infinite traces

The definition of contracts implicitly includes the crucial requirement that all breaches of contract are associated with a point in time. From this restriction it follows that contract conformance is not a *liveness property* [1], such as: Buyer must deliver the printer to Seller eventually. We see this as a natural restriction, since one of the purposes of formalising contracts is to run-time monitor their execution, and hence breaches of contract should be detected in finite time—in other words, every obligation must have a deadline.

The following lemma follows directly from the definition of contracts, because σ_τ is the longest prefix up to time τ of the trace σ.

Lemma 1. *Let* $\mathsf{c} : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ *be a contract and let* σ *be a (finite or infinite) trace. Then* $\mathsf{c}(\sigma) = (\tau, B)$ *if and only if* $\mathsf{c}(\sigma_\tau) = (\tau, B)$.

The previous lemma entails that any nonconforming trace (in particular, any nonconforming infinite trace) has a nonconforming prefix. However, not all extensions of this prefix need be nonconforming too. Indeed, a nonconforming finite trace may be extended to a conforming trace (for instance, simply by performing an unfulfilled obligation), even if the time of the breach coincides with the timestamp of the last event: a contract c may satisfy, for example, $\mathsf{c}(\langle(\tau, \alpha)\rangle) = (\tau, B)$ and $\mathsf{c}(\langle(\tau, \alpha), (\tau, \alpha')\rangle) = \checkmark$, for some $\alpha, \alpha' \in \mathsf{A}$, $\tau \in \mathsf{Ts}$, and parties $B \subseteq \mathsf{P}$. Still, any extension of a nonconforming finite trace *after* the time of the breach is also nonconforming.

Proposition 2. *The set of infinite traces conforming with a contract is a safety property.*

Proof. Let $\mathsf{c} : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be a contract and let

$$C = \{\sigma \in \mathsf{Tr}^{\tau_0} \mid \sigma \text{ is infinite and } \mathsf{c}(\sigma) = \checkmark\}.$$

We need to show that for any infinite trace $\sigma \notin C$, there is a prefix σ' of σ such that for any infinite trace σ'' with $\sigma' \sqsubset \sigma''$, it holds that $\sigma'' \notin C$.

Let $\sigma \notin C$ be an infinite trace. Then $\mathsf{c}(\sigma) = (\tau, B)$ for some τ and B. Let σ' be an arbitrary prefix of σ with $\mathrm{end}(\sigma') > \tau$, and consider an infinite trace σ'' with $\sigma' \sqsubset \sigma''$. Then, since $\mathrm{end}(\sigma') > \tau$, it follows that $\sigma''_\tau = \sigma_\tau$, and consequently condition (2) yields that $\mathsf{c}(\sigma'') = (\tau, B)$, hence $\sigma'' \notin C$, as required. □

The following lemma shows that "contracts" defined only on finite traces extend uniquely to contracts. In other words, contracts are uniquely determined by their verdicts on finite traces.

Lemma 3. *Let* P *be a set of parties and* $\mathsf{c} : \mathsf{Tr}^{\tau_0}_{\mathrm{fin}} \to \mathsf{V}$ *be a function such that if* $\mathsf{c}(\sigma) = (\tau, B)$ *then* $B \subseteq P$, $\tau \geq \tau_0$, *and* $\mathsf{c}(\sigma') = (\tau, B)$, *for all* $\sigma' \in \mathsf{Tr}^{\tau_0}_{\mathrm{fin}}$ *with* $\sigma_\tau = \sigma'_\tau$. *Then there exists a unique extension* $\mathsf{c}' : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ *of* c, *that is* $\mathsf{c} = \mathsf{c}'|_{\mathsf{Tr}^{\tau_0}_{\mathrm{fin}}}$, *such that* c' *is a contract.*

Proof. Let $\mathsf{c}' : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be the function that extends c to infinite traces by

$$\mathsf{c}'(\sigma) = \begin{cases} \checkmark & \text{if whenever } \mathsf{c}(\sigma') = (\tau, B) \text{ and } \sigma' \sqsubset \sigma \text{ then } \mathrm{end}(\sigma') \leq \tau, \\ \mathsf{c}(\sigma') & \text{otherwise, where } \sigma' \text{ is the shortest prefix of } \sigma \text{ such that } \mathsf{c}(\sigma') = (\tau', B') \text{ and } \mathrm{end}(\sigma') > \tau', \end{cases}$$

for any infinite trace σ. We first show that c' is a contract between parties P starting at time τ_0.

First note that $\mathsf{c}'(\sigma) = (\tau, B)$ if and only if there is $\sigma' \sqsubset \sigma$ with $\mathsf{c}(\sigma') = (\tau, B)$ and $\mathrm{end}(\sigma') > \tau$, hence property (1) follows immediately.

We show property (2), namely that if $\mathsf{c}'(\sigma) = (\tau, B)$ for some (finite or infinite) trace and some breach (τ, B), then $\mathsf{c}'(\sigma') = (\tau, B)$, for any (finite or infinite) trace σ' with $\sigma'_\tau = \sigma_\tau$. We can have one of the following cases:

- σ is finite and σ' is finite. This case follows directly from the hypotheses of the lemma.
- σ is finite and σ' is infinite. Then $\mathsf{c}'(\sigma) = \mathsf{c}(\sigma) = \mathsf{c}(\sigma_\tau)$. Let ϵ be such that $\sigma'_\tau\epsilon \sqsubset \sigma'$. We have $\mathrm{ts}(\epsilon) > \tau$, hence $\mathrm{end}(\sigma'_\tau\epsilon) > \tau$. Moreover, $\mathsf{c}(\sigma'_\tau\epsilon) = (\tau, B)$ as $(\sigma'_\tau\epsilon)_\tau = \sigma_\tau$. Hence, by definition, $\mathsf{c}'(\sigma') = (\tau, B)$.
- σ is infinite and σ' is finite. By definition of c', there is $\sigma'' \sqsubset \sigma$ such that $\mathsf{c}(\sigma'') = (\tau, B)$ and $\mathrm{end}(\sigma'') > \tau$. Then $\mathsf{c}(\sigma''_\tau) = (\tau, B)$. As $\sigma'_\tau = \sigma''_\tau$, it follows that $\mathsf{c}(\sigma') = (\tau, B)$.
- σ is infinite and σ' is infinite. As in the previous case, there is $\sigma'' \sqsubset \sigma$ such that $\mathsf{c}(\sigma''_\tau) = (\tau, B)$ and $\mathrm{end}(\sigma'') > \tau$. Then $\sigma''_\tau = \sigma_\tau = \sigma'_\tau$. Let ϵ be such that $\sigma'_\tau\epsilon \sqsubset \sigma'$. As in the second case, we obtain that $\mathsf{c}'(\sigma') = \mathsf{c}(\sigma''_\tau) = (\tau, B)$.

This shows that c' is a contract between parties P starting at time τ_0. We now prove that this extension is unique. Let c'' be a contract such that $\mathsf{c}''|_{\mathsf{Tr}^{\tau_0}_{\mathrm{fin}}} = \mathsf{c}$. We show that $\mathsf{c}' = \mathsf{c}''$. The contracts c' and c'' agree on all finite traces by construction, so assume for the sake of contradiction that $\mathsf{c}'(\sigma) \neq \mathsf{c}''(\sigma)$ for some infinite trace σ. Then either $\mathsf{c}'(\sigma) = (\tau, B)$ or $\mathsf{c}''(\sigma) = (\tau, B)$, for some τ and B, so assume that $\mathsf{c}'(\sigma) = (\tau, B)$. Then by Lemma 1 we have that $\mathsf{c}'(\sigma_\tau) = (\tau, B)$, and since σ_τ is finite, also $\mathsf{c}''(\sigma_\tau) = (\tau, B)$, and hence again by Lemma 1 we have that $\mathsf{c}''(\sigma) = (\tau, B)$, which is a contradiction. The case where $\mathsf{c}''(\sigma) = (\tau, B)$ is symmetric. □

2.4. Contract composition

By composing contracts, through conjunction and disjunction, new contracts are obtained. Given that a contract assigns verdicts to traces, defining such compositions amounts to stating how verdicts are composed.

Contract conjunction: This type of composition models the simultaneous commitment to several (sub)contracts. Conjunction is implicit in paper contracts: typically the involved parties have to conform with all the clauses therein. When some parties do not conform with some clauses, the resolution of blame assignment is given by the fundamental breach assumption: the earliest breach represents the overall verdict. When breaches of several clauses happen at the same time, then all breaching parties are to be blamed.

Definition 2. Let $\nu_1, \nu_2 \in \mathsf{V}$ be two verdicts. The *verdict conjunction* $\nu_1 \wedge \nu_2$ of ν_1 and ν_2 is given by:

$$\nu_1 \wedge \nu_2 = \begin{cases} \nu_1 & \text{if either } \nu_2 = \checkmark, \\ & \quad \text{or } \nu_1 = (\tau_1, B_1), \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 < \tau_2, \\ \nu_2 & \text{if either } \nu_1 = \checkmark, \\ & \quad \text{or } \nu_1 = (\tau_1, B_1), \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 > \tau_2, \\ (\tau, B) & \text{if } \nu_1 = (\tau, B_1), \nu_2 = (\tau, B_2), \text{ and } B = B_1 \cup B_2. \end{cases}$$

Definition 3. Let $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ and $\mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be two contracts. The *conjunction* of contracts is defined by

$$(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = \mathsf{c}_1(\sigma) \wedge \mathsf{c}_2(\sigma).$$

Note that $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = \checkmark$ if and only if $\mathsf{c}_1(\sigma) = \mathsf{c}_2(\sigma) = \checkmark$, for any trace σ.

The following lemma confirms the intuition that the conjunction of two contracts is a contract.

Lemma 4. *Let $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ and $\mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be two contracts between parties P_1 and P_2, respectively. Then the composition $\mathsf{c}_1 \wedge \mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ is a contract between parties $P_1 \cup P_2$.*

Proof. Property (1) follows immediately from the definition of verdict conjunction, so we need to prove property (2). Suppose that $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = (\tau, B)$ and σ' is such that $\sigma'_\tau = \sigma_\tau$. We can have one of the following cases:

- $\mathsf{c}_1(\sigma) = \checkmark$. Then $\mathsf{c}_2(\sigma) = (\tau, B)$ and it follows that $\mathsf{c}_2(\sigma') = (\tau, B)$.
  If $\mathsf{c}_1(\sigma') = \checkmark$ then clearly $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma') = (\tau, B)$. Suppose that $\mathsf{c}_1(\sigma') = (\tau', B')$ for some $(\tau', B') \neq (\tau, B)$. If $\tau' \leq \tau$ then $\sigma'_{\tau'} \sqsubset \sigma'_\tau \sqsubset \sigma$ and hence $\mathsf{c}_1(\sigma) = (\tau', B')$—contradiction. Hence $\tau' > \tau$. Since $(\tau, B) \wedge (\tau', B') = (\tau, B)$ it follows that $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = (\tau, B)$.
- $\mathsf{c}_2(\sigma) = \checkmark$. This case is symmetric to the previous one.
- $\mathsf{c}_1(\sigma) = (\tau_1, B_1)$ and $\mathsf{c}_2(\sigma) = (\tau_2, B_2)$ such that $(\tau_1, B_1) \wedge (\tau_2, B_2) = (\tau, B)$. We then have $\mathsf{c}_1(\sigma') = (\tau_1, B_1)$ and $\mathsf{c}_1(\sigma') = (\tau_2, B_2)$. Hence $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma') = (\tau, B)$. □

Example 2. Continuing Example 1, the first part of Paragraph 3 in Fig. 1 (that is "Buyer agrees to pay for the goods half upon receipt") can be represented by the contract c_3 between {Buyer}, where

$$\mathsf{c}_3(\sigma) = \begin{cases} \checkmark & \text{if } D = \emptyset, \text{ or if } D \neq \emptyset \text{ and } \sigma[j] = (\tau_1, \text{payment}_1) \text{ for some } j \text{ with } i_1 < j < |\sigma|, \\ (\tau_1, \{\text{Buyer}\}) & \text{otherwise,} \end{cases}$$

with $D = \{i \mid \sigma[i] = (\tau, \text{delivery}), 0 \leq i < |\sigma|, \tau \leq \tau_d\}$, $i_1 = \min(D)$, and $\tau_1 = \text{ts}(\sigma[i_1])$. Furthermore, the action payment_1 represents the first half payment to the Seller, and i_1 (τ_1) is the index (timestamp) that represents the receipt time of the first delivery, assuming that delivery time and receipt time coincide.

The second part of Paragraph 3 (that is "Buyer agrees to pay [...] the remainder within 30 days of delivery") can be encoded by the contract c'_3 between {Buyer}, where

$$\mathsf{c}'_3(\sigma) = \begin{cases} \checkmark & \text{if } D = \emptyset, \text{ or if } D \neq \emptyset \text{ and } \sigma[j] = (\tau, \text{payment}_2) \text{ for some } i_1 < j < |\sigma| \text{ and } \tau \leq \tau'_1, \\ (\tau'_1, \{\text{Buyer}\}) & \text{otherwise,} \end{cases}$$

with $\tau_1' = \tau_1 + 30$ (we assume that the time unit is 1 day), and the action payment$_2$ represents the second half payment to the Seller.

Using the previous lemma, Paragraph 3 of Fig. 1 is represented by the contract $\mathsf{c}_3 \wedge \mathsf{c}_3'$ between {Buyer}.

Contract disjunction: This type of composition models the situation where fulfilling only one of the clauses of a contract is sufficient to fulfil the entire contract. Unlike conjunction, the case when all clauses are breached is problematic, as each of the clauses is individually an option. To be able to give an answer in this case, we take a global view: all involved parties are at any time aware of the contract execution status. Thus, those parties responsible for the latest breach are to blame for the overall failure, because they should have fulfilled their obligations after knowing that other options are not available anymore. Still, when breaches happen at the same time, there is no other way than to choose nondeterministically between the breaches. Note that blaming the parties altogether is not a better alternative, as then the nondeterminism would be hidden somewhere else: the cause of the overall failure could be any of the causes of the individual breaches.

It is not a surprise that the treatment of disjunction is more complicated, since disjunction is inherently nondeterministic. Nevertheless, in the special case where all clauses stipulate commitments on the same contract participant, disjunction corresponds to a *choice* that said participant has. In this case it is clear who is to blame when all clauses are breached.

Definition 4. Let $\nu_1, \nu_2 \in \mathsf{V}$ be two verdicts such that if $\nu_1 = (\tau, B_1)$ and $\nu_2 = (\tau, B_2)$ then $B_1 = B_2$. The *verdict disjunction* $\nu_1 \vee \nu_2$ of ν_1 and ν_2 is given by:

$$\nu_1 \vee \nu_2 = \begin{cases} \checkmark & \text{if } \nu_1 = \checkmark \text{ or } \nu_2 = \checkmark, \\ (\tau_1, B_1) & \text{if } \nu_1 = (\tau_1, B_1),\ \nu_2 = (\tau_2, B_2) \text{ and } \tau_1 > \tau_2, \\ (\tau_2, B_2) & \text{if } \nu_1 = (\tau_1, B_1),\ \nu_2 = (\tau_2, B_2) \text{ and } \tau_1 < \tau_2, \\ (\tau, B) & \text{if } \nu_1 = \nu_2 = (\tau, B). \end{cases}$$

Two contracts c_1 and c_2 have *unique blame assignment* if for all traces σ, whenever $\mathsf{c}_1(\sigma) = (\tau, B_1)$ and $\mathsf{c}_2(\sigma) = (\tau, B_2)$, then $B_1 = B_2$.

Definition 5. Let $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ and $\mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be two contracts with unique blame assignment. The *disjunction* of contracts c_1 and c_2 is defined by

$$(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = \mathsf{c}_1(\sigma) \vee \mathsf{c}_2(\sigma).$$

Note that $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = \checkmark$ if and only if $\mathsf{c}_1(\sigma) = \checkmark$ or $\mathsf{c}_2(\sigma) = \checkmark$, for any $\sigma \in \mathsf{Tr}^{\tau_0}$.

The following lemma confirms the intuition that the disjunction of two contracts is a contract.

Lemma 5. *Let $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ and $\mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be two contracts with unique blame assignment, between parties P_1 and P_2, respectively. Then the composition $\mathsf{c}_1 \vee \mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ is a contract between parties $P_1 \cup P_2$.*

Proof. Property (1) follows immediately from the definition of verdict disjunction, so we need to prove property (2). Suppose that $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = (\tau, B)$ and σ' is such that $\sigma'_\tau = \sigma_\tau$. We can have one of the following cases:

- $\mathsf{c}_1(\sigma) = (\tau, B)$ and $\mathsf{c}_2(\sigma) = (\tau_2, B_2)$ with $\tau_2 < \tau$. It follows that $\mathsf{c}_1(\sigma') = (\tau, B)$ and $\mathsf{c}_2(\sigma_{\tau_2}) = (\tau_2, B_2)$. As $\sigma_{\tau_2} \sqsubset \sigma_\tau \sqsubset \sigma'$, we have $\mathsf{c}_2(\sigma') = (\tau_2, B_2)$. Hence $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = (\tau, B)$.
- $\mathsf{c}_2(\sigma) = (\tau, B)$ and $\mathsf{c}_1(\sigma) = (\tau_1, B_1)$ with $\tau_1 < \tau$. This case is symmetric to the previous one.
- $\mathsf{c}_1(\sigma) = (\tau, B)$ and $\mathsf{c}_2(\sigma) = (\tau, B)$. We then have $\mathsf{c}_1(\sigma') = (\tau, B)$ and $\mathsf{c}_1(\sigma') = (\tau, B)$. Hence $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma') = (\tau, B)$. □

Example 3. Continuing Example 2, the second part of Paragraph 4 in Fig. 1 (that is "an additional fine of 10% has to be paid within 14 days") can be encoded by the contract c_4 between {Buyer}:

$$\mathsf{c}_4(\sigma) = \begin{cases} \checkmark & \text{if } D = \emptyset, \text{ or if } D \neq \emptyset \text{ and } \sigma[j] = (\tau, \text{payment}_2') \text{ for some } i_1 < j < |\sigma| \text{ and } \tau \leq \tau_1'', \\ (\tau_1'', \{\text{Buyer}\}) & \text{otherwise,} \end{cases}$$

where $\tau_1'' = \tau_1 + 44$ and the action payment$_2'$ represents the payment of the second half together with the 10% fine by Buyer. (Note that the confusion with regard to the reference for the 10% computation would have to be solved at a different level—when defining payment$_2'$ concretely.)

As, for all traces, the contracts c_3' and c_4 only blame Buyer, the previous lemma ensures that $\mathsf{c}_3' \vee \mathsf{c}_4$ is a well-defined contract. The first four paragraphs are thus represented by the contract $\mathsf{c}_1 \wedge (\mathsf{c}_3 \wedge (\mathsf{c}_3' \vee \mathsf{c}_4))$ between {Buyer, Seller}. (We note that Paragraph 2 of Fig. 1 is encoded implicitly in the encoding of the other paragraphs.)

Algebraic properties of contract composition: The following lemma shows that the conjunction and disjunction operators on verdicts enjoy the expected algebraic properties, like commutativity, associativity, and distributivity.

Lemma 6. *Let $\nu, \nu_1, \nu_2, \nu_3, \nu'_1, \nu'_2, \nu'_3$ be verdicts such that if $\nu'_i = (\tau, B_i)$ and $\nu'_j = (\tau, B_j)$ then $B_i = B_j$, for any $i, j \in \{1, 2, 3\}$. Then the following equalities hold:*

$$\nu_1 \wedge \nu_2 = \nu_2 \wedge \nu_1 \qquad \text{(Commutativity)}$$
$$\nu'_1 \vee \nu'_2 = \nu'_2 \vee \nu'_1 \qquad \text{(Commutativity)}$$
$$\nu_1 \wedge (\nu_2 \wedge \nu_3) = (\nu_1 \wedge \nu_2) \wedge \nu_3 \qquad \text{(Associativity)}$$
$$\nu'_1 \vee (\nu'_2 \vee \nu'_3) = (\nu'_1 \vee \nu'_2) \vee \nu'_3 \qquad \text{(Associativity)}$$
$$\nu'_1 \vee (\nu'_1 \wedge \nu'_2) = \nu'_1 \qquad \text{(Absorption)}$$
$$\nu'_1 \wedge (\nu'_1 \vee \nu'_2) = \nu'_1 \qquad \text{(Absorption)}$$
$$\nu'_1 \vee (\nu'_2 \wedge \nu'_3) = (\nu'_1 \vee \nu'_2) \wedge (\nu'_1 \vee \nu'_3) \qquad \text{(Distributivity)}$$
$$\nu_1 \wedge (\nu'_2 \vee \nu'_3) = (\nu_1 \wedge \nu'_2) \vee (\nu_1 \wedge \nu'_3) \qquad \text{(Distributivity)}$$
$$\checkmark \wedge \nu = \nu \wedge \checkmark = \nu \qquad \text{(Unit)}$$
$$\checkmark \vee \nu = \nu \vee \checkmark = \checkmark \qquad \text{(Unit)}$$

Proof. These equalities follow directly from Definitions 2 and 4. □

These algebraic properties are easily lifted from verdicts to contracts, which allows us to perform algebraic, meaning-preserving rewritings of contracts.

Corollary 7. *Let C be a set of contracts that is closed under contract conjunction and disjunction, $\mathsf{c}_\checkmark \in C$, and for all $\mathsf{c}_1, \mathsf{c}_2 \in C$, the contracts c_1 and c_2 have unique blame assignment. Then $(C, \vee, \wedge)$ is a distributive lattice with unit element $\mathsf{c}_\checkmark$.*

We recall that the idempotency equalities $\mathsf{c} \wedge \mathsf{c} = \mathsf{c}$ and $\mathsf{c} \vee \mathsf{c} = \mathsf{c}$, that hold for any contract c, follow from the absorption equalities. We also note that the equalities that only concern conjunction hold for arbitrary contracts.

2.5. Run-time monitoring

The contract model presented above considers complete traces, which are either finite or infinite, and there is no restriction as to whether the verdict of a contract can be computed or not. For run-time monitoring, however, traces are always partial and finite, and it should be possible to compute verdicts at run-time. We consequently define, abstractly, what constitutes run-time monitoring for the contract model, using a conventional many-valued semantics [13].

The output of a run-time monitor is an element of the union of the sets $\mathsf{V}_\star = \{\nu_\star \mid \nu \in \mathsf{V}\}$ for $\star \in \{!, ?\}$, where $\nu_!$ is a *final verdict*, and $\nu_?$ is a *potential verdict*. Final verdicts are output when all extensions of the current partial trace have the same verdict. In other words, the verdict on the complete trace, whatever this would be, is uniquely determined by (the verdict on) the partial trace; there is hence no need to perform further monitoring. In contrast, potential verdicts are output when the verdicts on extensions of the current partial trace differ. Of course, if the current trace is a complete trace (in this case no more events occur), then the potential verdict is the actual verdict on this trace.

Definition 6. Let $\mathsf{c} : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be a contract between parties P. A *run-time monitor* for c is a *computable* function $\mathrm{mon} : \mathsf{Tr}^{\tau_0}_{\mathrm{fin}} \to \mathsf{V}_! \cup \mathsf{V}_?$ that satisfies

$$\mathrm{mon}(\sigma) = \begin{cases} \checkmark_! & \text{if } \mathsf{c}(\sigma') = \checkmark \text{ for all } \sigma' \text{ with } \sigma \sqsubset \sigma', \\ (\tau, B)_! & \text{if } \mathsf{c}(\sigma') = (\tau, B) \text{ for all } \sigma' \text{ with } \sigma \sqsubset \sigma', \\ \checkmark_? & \text{if } \mathsf{c}(\sigma) = \checkmark \text{ and } \mathsf{c}(\sigma') \neq \checkmark \text{ for some } \sigma \sqsubset \sigma', \\ (\tau, B)_? & \text{if } \mathsf{c}(\sigma) = (\tau, B) \text{ and } \mathsf{c}(\sigma') \neq (\tau, B) \text{ for some } \sigma \sqsubset \sigma'. \end{cases}$$

Note that, in case of a potential breach, that is if $\mathrm{mon}(\sigma) = (\tau, B)_?$ then condition (2) of Definition 1 guarantees that $\mathrm{end}(\sigma) \leq \tau$, hence $(\tau, B)_?$ is always an indication of a future—but avoidable—breach.

The definition expresses both *impartiality* and *anticipation* [13]. Impartiality means that a final verdict is only output if the partial trace cannot be extended into a complete trace with a different verdict. Formally,

$$\text{if } \mathrm{mon}(\sigma) = \nu_! \text{ then } \mathsf{c}(\sigma') = \nu \text{ for all } \sigma' \text{ with } \sigma \sqsubset \sigma'.$$

$$
\begin{array}{llr}
s ::= & \textbf{letrec } \{f_i(\vec{x}_i)\langle\vec{y}_i\rangle = c_i\}_{i=1}^n \textbf{ in } c \textbf{ starting } \tau & \text{(CSL specification)} \\
\\
c ::= & \textbf{fulfilment} & \text{(No obligations)} \\
 & \mid \langle e_1\rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 & \text{(Obligation)} \\
 & \mid \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 & \text{(External choice)} \\
 & \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 & \text{(Internal choice)} \\
 & \mid c_1 \textbf{ and } c_2 & \text{(Conjunction)} \\
 & \mid c_1 \textbf{ or } c_2 & \text{(Disjunction)} \\
 & \mid f(\vec{e}_1)\langle\vec{e}_2\rangle & \text{(Instantiation)} \\
\\
e ::= & x \mid v \mid \neg e_1 \mid e_1 \star e_2 \mid e_1 \prec e_2 & \text{(Expression)} \\
d ::= & \textbf{after } e_1 \textbf{ within } e_2 & \text{(Deadline expression)}
\end{array}
$$

Fig. 2. The grammar of CSL. $f \in \mathcal{F}$ ranges over template names, $x, y, z \in \mathcal{V}$ range over variables, $k \in \mathcal{K}$ ranges over action kinds, and $v \in \bigcup_{t\in\mathcal{T}} [\![t]\!]$ ranges over values. Furthermore, $\star \in \{+, -, *, /, \wedge\}$ and $\prec \in \{<, =\}$.

Anticipation is the reverse of impartiality. It means that inevitable—possibly future—verdicts are output as early as possible, that is a potential verdict is only output if it is possible to reach a different verdict. Formally,

$$\text{if } \mathsf{c}(\sigma') = \nu \text{ for all } \sigma' \text{ with } \sigma \sqsubset \sigma' \text{ then } \mathrm{mon}(\sigma) = \nu_!.$$

Anticipation can be relaxed, for instance by allowing final breaches to be output only when the time of breach has been reached, but impartiality is a crucial requirement for run-time monitoring which cannot be relaxed.

Example 4. Consider the contract $\mathsf{c}_1 \wedge (\mathsf{c}_3 \wedge (\mathsf{c}_3' \vee \mathsf{c}_4))$ between {Buyer, Seller} from Example 3, and the following events:

$$\begin{aligned}
&\epsilon_1 = (\text{2011-01-01, delivery}), && \epsilon_2 = (\text{2011-01-02, delivery}),\\
&\epsilon_3 = (\text{2011-01-01, payment}_1), && \epsilon_4 = (\text{2011-01-10, payment}_2),\\
&\epsilon_5 = (\text{2011-02-10, payment}_2').
\end{aligned}$$

The output of an associated run-time monitor on the following sample traces is as follows:

$$\begin{aligned}
\mathrm{mon}(\langle\rangle) &= (\text{2011-01-01, \{Seller\}})_?,\\
\mathrm{mon}(\langle\epsilon_2\rangle) &= (\text{2011-01-01, \{Seller\}})_!,\\
\mathrm{mon}(\langle\epsilon_1\rangle) &= (\text{2011-01-01, \{Buyer\}})_?,\\
\mathrm{mon}(\langle\epsilon_1, \epsilon_3\rangle) &= (\text{2011-02-14, \{Buyer\}})_?,\\
\mathrm{mon}(\langle\epsilon_1, \epsilon_3, \epsilon_4\rangle) = \mathrm{mon}(\langle\epsilon_1, \epsilon_3, \epsilon_5\rangle) &= \checkmark_!.
\end{aligned}$$

3. A contract specification language

The previous section provided a semantic account for compositional contracts. However, it is cumbersome to specify contracts directly in the abstract model, as we have seen in Examples 1–3. Thus we propose a contract specification language, CSL, which enables succinct, syntactic representation of real-world contracts in a human-readable form, and which has a formal semantics in terms of the abstract contract model. The primary target of CSL is business contracts, but rather than fixing the set of actions to for instance payments and deliveries, we parametrise the language with respect to a signature, which can be thought of as the vocabulary used in a contract.

Formally, a *signature* is a triple $S = (\mathcal{K}, \mathrm{ar}, \mathcal{T})$, where $\mathcal{K}$ is a finite set of *action kinds* with associated *arities* and *types*, $\mathrm{ar} : \mathcal{K} \to \mathcal{T}^*$, where $\mathcal{T}$ is a finite set of types. The *domain* of a type t is denoted by $[\![t]\!]$, and we assume that $\mathcal{T}$ contains the basic types Bool, Int, Time, and Party, with the corresponding domains $[\![\text{Bool}]\!] = \{\textbf{false}, \textbf{true}\}$, $[\![\text{Int}]\!] = \mathbb{Z}$, $[\![\text{Time}]\!] = \mathsf{Ts}$, and $[\![\text{Party}]\!] = \mathsf{P}$, respectively. Signatures provide structure to actions, and we consequently redefine the set of actions, with respect to a given signature, as follows: $\mathsf{A} = \{k(\vec{v}) \mid k \in \mathcal{K}, \mathrm{ar}(k) = \langle t_1, \dots, t_n\rangle, \text{ and } \vec{v} \in [\![t_1]\!] \times \cdots \times [\![t_n]\!]\}$. Furthermore, we assume an infinite set of variables $\mathcal{V}$, ranged over by x, y, z, and an infinite set of template names $\mathcal{F}$, ranged over by f.

3.1. CSL syntax

The grammar of CSL is presented in Fig. 2. In what follows, we describe informally each construct of the language.

The atomic expressions of CSL are values $v \in [\![t]\!]$ of some type t and variables. From integer values and variables, arithmetic and Boolean *expressions* are formed by using arithmetic operators, equalities, and inequalities. We note in particular that “/”

denotes integer division and the specification needs to take into account the possible loss in precision with regard to real division. Abusing language, a *deadline expression* actually represents an interval of integers, as explained shortly.

A CSL *specification* s is a set of template definitions together with a body c and an absolute point in time τ, which defines the starting time of the contract. Templates can be instantiated in the body of the specification. Mutual recursion is allowed and it enables potentially infinite contract executions. The parameters of a template are values $\vec{x}$ and parties $\vec{y}$. Value parameters are dynamic, that is they can be instantiated with values from earlier events, whereas party parameters are static, that is all parties are fixed before the contract is started, and they do not change over time.

Clauses describe the normative content of contracts. The bodies of CSL specifications and of template definitions are clauses. All deadlines that occur in clauses are relative to unspecified reference points which are given by the starting time of the specification and by the time of event occurrences. Thus, these relative deadlines are only lifted to absolute deadlines when the CSL specification is executed. The only atomic clause is **fulfilment**, which represents the clause that is always fulfilled.

Fully instantiated obligation clauses have the form

$$\langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due after}\ n_1\ \textbf{within}\ n_2\ \textbf{remaining}\ z\ \textbf{then}\ c,$$

which should be read:

> Party p is responsible that (but need not be in charge of) an action of kind k satisfying condition e takes place. This action should happen after n_1 time units, but within n_2 time units thereafter. If these requirements are satisfied, then the *continuation clause* c determines any further obligations.

The variables of the vector $\vec{x}$ are bound to the parameters of the action, and their scope is e and c. The variable z is bound to the remainder of the deadline: if the deadline is for instance **after** 2 **within** 5 and the action takes place 4 time units after the reference point, then z is bound to $(2 + 5) - 4 = 3$. The scope of z is c only. All deadlines in the continuation c are relative to the time of the action.

External choices are similar to obligation clauses, but they contain an alternative continuation branch which becomes active if the deadline passes. For this reason, external choices have no responsible party parameter, since no one has to be blamed in case the deadline expires.

The clause **if** e **then** c_1 **else** c_2 represents an internal choice, where the branching condition e can be computed directly without having to wait for external input (that is for events). The clauses c_1 **and** c_2 and c_1 **or** c_2 represent clause conjunction and disjunction, respectively. Finally, $f(\vec{e}_1)\langle\vec{e}_2\rangle$ is instantiation of template f, where $\vec{e}_1$ are value parameters and $\vec{e}_2$ are party parameters.

We use standard syntactic sugar such as $e_1 \vee e_2$ for $\neg(\neg e_1 \wedge \neg e_2)$, $e_1 \leq e_2$ for $(e_1 < e_2) \vee (e_1 = e_2)$, and $e_1 \neq e_2$ for $\neg(e_1 = e_2)$. Also, we omit continuations and **else** branches if they are **fulfilment**, we omit the **after** part of a deadline if it is 0, we write **immediately** for **within** 0, and we omit the **remaining** part if it is not used. Finally, we use abbreviations like 30D to denote the value representing an amount of time of 30 days, that is the integer $30 * 24 * 60 * 60$, assuming that the time unit is of 1 s.

In terms of deontic modalities [28], it may seem that CSL only supports obligations, and not permissions and prohibitions. However, permissions in a contractual context are only of interest if they entail new obligations (on counter parties). Hence we model permissions as external choices that trigger new obligations, as illustrated in the following example. Prohibitions can also be modelled as external choices, where the consequence is an unfullfilable obligation on the party who performed the prohibited action, as we shall see in Section 3.7, where we provide further examples.

Example 5. Fig. 3 shows the specification in CSL of the sales contract in Fig. 1. The formalisation assumes a signature that includes the action kinds $\{\text{Deliver}, \text{Payment}, \text{Return}\} \subseteq \mathcal{K}$, with types $\text{ar}(\text{Deliver}) = \text{ar}(\text{Return}) = \langle\text{Party}, \text{Party}, \text{String}\rangle$ and $\text{ar}(\text{Payment}) = \langle\text{Party}, \text{Party}, \text{Int}\rangle$. The domain of String is the set of all strings, and the two parties of each action kind represent the sender and receiver, respectively. The example disambiguates the informal contract: the 10% fine is calculated with respect to half of the total price, and Buyer is only entitled to return the goods if the first half is paid upon delivery. A different disambiguation could be given by another CSL specification. Note also how we encode the permission to return the goods as an external choice which has the consequence that Seller has to pay the original amount back to Buyer.

3.2. CSL type system

We equip CSL with a type system. For this purpose, we define different *typing judgments* over an implicit signature $S = (\mathcal{K}, \text{ar}, \mathcal{T})$. Before presenting the typing judgments, we introduce some notation. We write $f : A \rightharpoonup_{\text{fin}} B$ for a partial function f from A to B with a finite domain. Furthermore, $f[a \mapsto b]$ denotes the function which maps a to b and behaves like f on all other input. We write $f[\vec{a} \mapsto \vec{b}]$ for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$, for vectors $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_n)$. Finally, we write $A \subseteq_{\text{fin}} B$ to say that $A \subseteq B$ and A is finite.

```
letrec sale(deliveryDeadline, goods, payment)⟨buyer, seller⟩ =
 ⟨seller⟩ Deliver(s,r,g) where s = seller ∧ r = buyer ∧ g = goods
                          due within deliveryDeadline
 then
 ⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = payment/2
                          due immediately
 then
 ((⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = payment/2
                          due within 30D
   or
   ⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = (payment/2) ∗ 110/100
                          due within 14D after 30D)
  and
  if Return(s,r,g) where s = buyer ∧ r = seller ∧ g = goods due within 14D then
    ⟨seller⟩ Payment(s,r,a) where s = seller ∧ r = buyer ∧ a = payment due within 7D)
in
sale(0, “Laser printer”, 200)⟨Buyer, Seller⟩ starting 2011-01-01
```

Fig. 3. A CSL specification of a sales contract between a buyer and a seller.

$$\boxed{\Gamma \vdash e : t} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{v \in [\![t]\!]}{\Gamma \vdash v : t}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash e_1 \star e_2 : \mathrm{Int}}\ (\star \in \{+, -, *\}) \qquad \frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad n_2 \in [\![\mathrm{Int}]\!]}{\Gamma \vdash e_1 / n_2 : \mathrm{Int}}\ (n_2 \neq 0)$$

$$\frac{\Gamma \vdash e : \mathrm{Bool}}{\Gamma \vdash \neg e : \mathrm{Bool}} \qquad \frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \mathrm{Bool}}{\Gamma \vdash e_1 \wedge e_2 : \mathrm{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash e_1 < e_2 : \mathrm{Bool}} \qquad \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \mathrm{Bool}}$$

$$\boxed{\Lambda \vdash e' : P} \qquad \frac{x \in \mathcal{V}}{\{x\} \vdash x : \emptyset} \qquad \frac{p \in \mathsf{P}}{\emptyset \vdash p : \{p\}}$$

$$\boxed{\Gamma \vdash d : \mathrm{Deadline}} \qquad \frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash \textbf{after}\ e_1\ \textbf{within}\ e_2 : \mathrm{Deadline}}$$

Fig. 4. Typing judgments for expressions e, party expressions e', and deadline expressions d.

Our typing judgments use the following typing environments:

$$\Lambda \subseteq_{\mathrm{fin}} \mathcal{V} \qquad \text{(Party typing environment)}$$
$$\Gamma : \mathcal{V} \rightharpoonup_{\mathrm{fin}} \mathcal{T} \qquad \text{(Variable typing environment)}$$
$$\Delta : \mathcal{F} \rightharpoonup_{\mathrm{fin}} \mathcal{T}^* \times \mathbb{N} \qquad \text{(Template typing environment)}$$

The typing environment for parties Λ keeps track of parametrised parties (such as the parameter *buyer* of the template *sale* in Fig. 3), and the typing environment for values Γ keeps track of parametrised values and their type (such as the parameter *goods* of the template *sale* in Fig. 3). The typing environment for clause templates Δ associates with each template name the types of its parameters and the number of party parameters. Also, we use the meta-types Deadline, Clause$\langle P\rangle$, and Contract$\langle P\rangle$, parametrised by a finite set of parties $P \subseteq_{\mathrm{fin}} \mathsf{P}$, to represent the type of deadlines, clauses involving parties P, and contracts involving parties P, respectively.

The typing judgments for expressions $\Gamma \vdash e : t$, for party expressions (that is the expressions determining responsibility in obligations) $\Lambda \vdash e' : P$, and for deadline expressions $\Gamma \vdash d :$ Deadline are presented in Fig. 4. The typing rules for expressions are standard, but we require that the denominator of a division expression be known statically in order to avoid division by zero. The typing rules for party expressions $\Lambda \vdash e' : P$ are used to determine the parties that are involved in a given clause.

The typing rules for clauses $\Delta, \Lambda, \Gamma \vdash c$: Clause$\langle P\rangle$, for template definitions $\Delta \vdash D$, and for full CSL specifications $\vdash s$: Contract$\langle P\rangle$ are presented in Fig. 5. A derivation $\Delta, \Lambda, \Gamma \vdash c$: Clause$\langle P\rangle$ intuitively means that in template environment Δ and variable environment Γ, c is a clause in which only parties P and parametrised parties Λ can be blamed for a breach of contract. The typing rule for clause disjunction, c_1 **or** c_2, uses this invariant to check that at most one party can breach either c_1 or c_2, which guarantees that verdict disjunction is well-defined. The typing rules for obligations and external choices illustrate the scope of the bound variables $\vec{x}$ and z.

$$\boxed{\Delta, \Lambda, \Gamma \vdash c : \mathrm{Clause}\langle P \rangle} \qquad \frac{}{\Delta, \emptyset, \Gamma \vdash \textbf{fulfilment} : \mathrm{Clause}\langle \emptyset \rangle}$$

$$\frac{\begin{array}{c} \Gamma' = \Gamma[\vec{x} \mapsto \mathrm{ar}(k)] \\ \Gamma_2 = \Gamma'[z \mapsto \mathrm{Int}] \end{array} \qquad \begin{array}{c} \Lambda_1 \vdash e_1 : P_1 \\ \Gamma' \vdash e : \mathrm{Bool} \\ \Gamma \vdash d : \mathrm{Deadline} \end{array} \qquad \Delta, \Lambda_2, \Gamma_2 \vdash c : \mathrm{Clause}\langle P_2 \rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash \langle e_1 \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c : \mathrm{Clause}\langle P_1 \cup P_2 \rangle}$$

$$\frac{\begin{array}{c} \Gamma' = \Gamma[\vec{x} \mapsto \mathrm{ar}(k)] \\ \Gamma_1 = \Gamma'[z \mapsto \mathrm{Int}] \end{array} \qquad \begin{array}{c} \Gamma' \vdash e : \mathrm{Bool} \\ \Gamma \vdash d : \mathrm{Deadline} \end{array} \qquad \begin{array}{c} \Delta, \Lambda_1, \Gamma_1 \vdash c_1 : \mathrm{Clause}\langle P_1 \rangle \\ \Delta, \Lambda_2, \Gamma \vdash c_2 : \mathrm{Clause}\langle P_2 \rangle \end{array}}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \mathrm{Clause}\langle P_1 \cup P_2 \rangle}$$

$$\frac{\Gamma \vdash e : \mathrm{Bool} \qquad \Delta, \Lambda_1, \Gamma \vdash c_1 : \mathrm{Clause}\langle P_1 \rangle \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 : \mathrm{Clause}\langle P_2 \rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \mathrm{Clause}\langle P_1 \cup P_2 \rangle}$$

$$\frac{\Delta, \Lambda_1, \Gamma \vdash c_1 : \mathrm{Clause}\langle P_1 \rangle \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 : \mathrm{Clause}\langle P_2 \rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash c_1\ \textbf{and}\ c_2 : \mathrm{Clause}\langle P_1 \cup P_2 \rangle}$$

$$\frac{|\Lambda_1 \cup \Lambda_2| + |P_1 \cup P_2| \leq 1 \qquad \Delta, \Lambda_1, \Gamma \vdash c_1 : \mathrm{Clause}\langle P_1 \rangle \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 : \mathrm{Clause}\langle P_2 \rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash c_1\ \textbf{or}\ c_2 : \mathrm{Clause}\langle P_1 \cup P_2 \rangle}$$

$$\frac{\Delta(f) = (\langle t_1, \ldots, t_m \rangle, n) \qquad \forall i \in \{1, \ldots, m\}.\, \Gamma \vdash e_i : t_i \qquad \forall i \in \{1, \ldots, n\}.\, \Lambda_i \vdash e'_i : P_i}{\Delta, \bigcup_{i=1}^{n} \Lambda_i, \Gamma \vdash f(e_1, \ldots, e_m)\langle e'_1, \ldots, e'_n \rangle : \mathrm{Clause}\langle \bigcup_{i=1}^{n} P_i \rangle}$$

$$\boxed{\Delta \vdash D} \quad \frac{\begin{array}{c} \Gamma_i = \left[\vec{x}_i \mapsto \vec{t}_i, \vec{y}_i \mapsto \overrightarrow{\mathrm{Party}}\right] \\ \forall i, j \in \{1, \ldots, n\}. i \neq j \Rightarrow f_i \neq f_j \\ \Delta = \left[f_1 \mapsto (\vec{t}_1, |\vec{y}_1|), \ldots, f_n \mapsto (\vec{t}_n, |\vec{y}_n|)\right] \end{array} \qquad \forall i \in \{1, \ldots, n\}.\, \Delta, \vec{y}_i, \Gamma_i \vdash c_i : \mathrm{Clause}\langle \emptyset \rangle}{\Delta \vdash \{f_i(\vec{x}_i)\langle \vec{y}_i \rangle = c_i\}_{i=1}^{n}}$$

$$\boxed{\vdash s : \mathrm{Contract}\langle P \rangle} \qquad \frac{\Delta \vdash D \qquad \Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P \rangle}{\vdash \textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau : \mathrm{Contract}\langle P \rangle}$$

Fig. 5. Typing judgments for CSL clauses *c*, template definitions *D*, and specifications *s*.

The typing rule for template definitions $\Delta \vdash D$ requires that the body of each definition contains no "hard coded" parties, that is it may only contain variables, but not values of type Party. The restriction is strictly speaking not necessary, however we consider it best practice not to have hard coded parties inside template definitions, and we therefore rule out this possibility. We furthermore allow party parameters to be used in the scope of ordinary expressions; see the definition of Γ_i, and the body of the template *sale* in Fig. 3 for an example.

An expression *e* is *well-typed* in the variable typing environment Γ, if there is a type *t* such that $\Gamma \vdash e : t$. Similarly, a deadline expression *d* is well-typed in the variable typing environment Γ, if $\Gamma \vdash d$: Deadline. A clause *c* involving parties *P* is well-typed in the variable environment Γ, party environment Λ, and template environment Δ, if $\Delta, \Lambda, \Gamma \vdash c : \mathrm{Clause}\langle P \rangle$. A specification *s* involving parties *P* is well-typed, if $\vdash s : \mathrm{Contract}\langle P \rangle$. We say simply that a CSL construct is well-typed, if there are appropriate environments and involved parties within which the construct is well-typed.

Lastly, we remark that the type system presented here is declarative, that is checking whether CSL specifications are well-typed cannot be directly implemented based on the given typing rules. This is because of the rule for template definitions, for which one has to guess the types of value parameters. An actual implementation will either rely on explicit type annotations of template parameters or perform type inference. While we treat neither approaches formally here, we note that explicit type annotations will immediately give rise to an algorithmic type system.

3.3. Well-formed specifications

Unfolding of template definitions need not always terminate—even for well-typed specifications—as illustrated in the following example:

$$s_\Omega = \textbf{letrec}\ f()\langle\rangle = f()\langle\rangle\ \textbf{in}\ f()\langle\rangle\ \textbf{starting}\ 2011\text{-}01\text{-}01$$

$$\boxed{e \Downarrow v} \quad \frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \star e_2 \Downarrow n_1 \star n_2} \ (\star \in \{+,-,*,/\}) \qquad \frac{e \Downarrow \textbf{true}}{\neg e \Downarrow \textbf{false}} \qquad \frac{e \Downarrow \textbf{false}}{\neg e \Downarrow \textbf{true}}$$

$$\frac{e_1 \Downarrow \textbf{true} \qquad e_2 \Downarrow \textbf{true}}{e_1 \wedge e_2 \Downarrow \textbf{true}} \qquad \frac{e_1 \Downarrow \textbf{false}}{e_1 \wedge e_2 \Downarrow \textbf{false}} \qquad \frac{e_2 \Downarrow \textbf{false}}{e_1 \wedge e_2 \Downarrow \textbf{false}}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \prec e_2 \Downarrow b} \ \left(\prec \in \{<,=\}, b = \begin{cases} \textbf{true}, & \text{if } v_1 \prec v_2 \\ \textbf{false}, & \text{if } v_1 \not\prec v_2 \end{cases} \right)$$

$$\boxed{d \Downarrow^{\tau} (\tau_1, \tau_2)} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{\textbf{after } e_1 \textbf{ within } e_2 \Downarrow^{\tau} (\tau + n_1, \tau + n_1 + n_2)}$$

Fig. 6. Evaluation of expressions and deadline expressions.

We avoid such ill-formed specifications by considering only specifications that satisfy a certain syntactic criterion that we introduce next.

Given a clause c, we recursively define the *immediate subclauses* of c as follows:

$$\mathrm{Sub}(c) = \{c\} \cup \begin{cases} \mathrm{Sub}(c_2) & \text{if } c = \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2, \\ \mathrm{Sub}(c_1) \cup \mathrm{Sub}(c_2) & \text{if } c = c_1 \textbf{ and } c_2, \\ \mathrm{Sub}(c_1) \cup \mathrm{Sub}(c_2) & \text{if } c = c_1 \textbf{ or } c_2, \\ \mathrm{Sub}(c_1) \cup \mathrm{Sub}(c_2) & \text{if } c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \\ \emptyset & \text{otherwise.} \end{cases}$$

Given a set of template definitions D, we let $\mathcal{F}_D$ denote the names of the templates defined in D. The *immediate unfolding relation* $\Rightarrow_D$ on $\mathcal{F}_D$ is defined as follows: $f \Rightarrow_D g$ if and only if there is a subclause $g(\vec{e}_1)\langle\vec{e}_2\rangle \in \mathrm{Sub}(c_f)$ where c_f is such that $(f(\vec{x})\langle\vec{y}\rangle = c_f) \in D$. Intuitively, $\Rightarrow_D$ represents a dependency relation between templates, where $f \Rightarrow_D g$ means that the unfolding of f requires an immediate unfolding of g. The definition of immediate subclauses reflects this intuition. For instance, in the continuation clause c_1 of an obligation, the templates in c_1 are not immediately instantiated—they are instantiated only after the obligation is fulfilled.

We say that a specification s is *well-formed* with parties P, if s involving parties P is well-typed and the immediate unfolding relation on the template names of s is acyclic. By requiring that the unfolding relation be acyclic, we avoid exactly those cases where the unfolding of a template f requires a series of immediate unfoldings leading to an unfolding of f itself. Note that the specification given in Fig. 3 is well-formed, while the above specification s_Ω is not.

3.4. CSL semantics

We now present the operational semantics for CSL, which is used to define the mapping of CSL specifications to abstract contracts, and which gives rise to a run-time monitoring algorithm as well. Inspired by Andersen et al. [2], we define a reduction semantics, which has the advantage that residual obligations, after an event has taken place, can be seen directly by inspecting the reduced term. More generally it follows that any analysis applicable to initial CSL specifications will also be applicable at any given point in time, since running CSL specifications are conceptually no different from initial specifications.

We first define the evaluation of well-typed expressions $e \Downarrow v$ and well-typed deadline expressions $d \Downarrow^{\tau} (\tau_1, \tau_2)$ in Fig. 6, using standard derivation rules. The timestamp τ in the rule for deadlines is the time with respect to which relative deadlines are calculated. It represents the starting time of the specification or the time of its last update, which equals the time of the last event occurrence. The following lemma shows the expected correspondence between the typing rules and the evaluation rules for (deadline) expressions.

Lemma 8. *Let e be an expression, d be a deadline expression, and t be a type. If $\emptyset \vdash e : t$, then there is a unique $v \in [\![t]\!]$ such that $e \Downarrow v$. If $\emptyset \vdash d$: Deadline, then for any $\tau \in \mathsf{Ts}$, there are unique $\tau_1, \tau_2 \in \mathbb{Z}$ with $d \Downarrow^{\tau} (\tau_1, \tau_2)$.*

Proof. For the first claim, existence follows by induction on the derivation of $\emptyset \vdash e : t$, while uniqueness follows by structural induction on e. The last claim follows immediately from the first one. □

During reductions, variables are instantiated with values in expressions and clauses. Since party parameters do not depend on event data, we use two kinds of (applications of) substitutions, namely substitutions of value parameters and substitutions of party parameters. Formally, a *(value) substitution* is an element of the set $\mathcal{V} \rightharpoonup_{\text{fin}} \bigcup_{t \in \mathcal{T}} [\![t]\!]$. A *party substitution* is a substitution having P as the codomain. Hence, party substitutions are special cases of value substitutions.

$$\boxed{e[\theta]}\qquad \begin{aligned} x[\theta] &= \begin{cases} \theta(x), & \text{if } x \in \mathrm{dom}(\theta) \\ x, & \text{otherwise} \end{cases} \\ v[\theta] &= v \\ (\neg e)[\theta] &= \neg e[\theta] \\ (e_1 \star e_2)[\theta] &= e_1[\theta] \star e_2[\theta] \\ (e_1 \prec e_2)[\theta] &= e_1[\theta] \prec e_2[\theta] \end{aligned}$$

$$\boxed{d[\theta]}\qquad (\textbf{after } e_1 \textbf{ within } e_2)[\theta] = \textbf{after } e_1[\theta] \textbf{ within } e_2[\theta]$$

$$\boxed{c[\theta]}\qquad \begin{aligned} \textbf{fulfilment}[\theta] &= \textbf{fulfilment} \\ \begin{pmatrix} \langle e_1\rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c \end{pmatrix}[\theta] &= \begin{matrix} \langle e_1\rangle\ k(\vec{x}) \textbf{ where } e_2[\theta|_{\mathcal{V}\setminus\vec{x}}] \textbf{ due } d[\theta] \\ \textbf{remaining } z \textbf{ then } c[\theta|_{\mathcal{V}\setminus(\vec{x}\cup\{z\})}] \end{matrix} \\ \begin{pmatrix} \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \end{pmatrix}[\theta] &= \begin{matrix} \textbf{if } k(\vec{x}) \textbf{ where } e[\theta|_{\mathcal{V}\setminus\vec{x}}] \textbf{ due } d[\theta] \\ \textbf{remaining } z \textbf{ then } c_1[\theta|_{\mathcal{V}\setminus(\vec{x}\cup\{z\})}] \textbf{ else } c_2[\theta] \end{matrix} \\ (c_1 \textbf{ and } c_2)[\theta] &= c_1[\theta] \textbf{ and } c_2[\theta] \\ (c_1 \textbf{ or } c_2)[\theta] &= c_1[\theta] \textbf{ or } c_2[\theta] \\ (\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2)[\theta] &= \textbf{if } e[\theta] \textbf{ then } c_1[\theta] \textbf{ else } c_2[\theta] \\ f(e_1, \dots, e_n)\langle\vec{e'}\rangle[\theta] &= f(e_1[\theta], \dots, e_n[\theta])\langle\vec{e'}\rangle \end{aligned}$$

$$\boxed{c\langle\theta\rangle}\qquad \begin{aligned} \textbf{fulfilment}\langle\theta\rangle &= \textbf{fulfilment} \\ \begin{pmatrix} \langle e_1\rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c \end{pmatrix}\langle\theta\rangle &= \begin{matrix} \langle e_1[\theta]\rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c\langle\theta\rangle \end{matrix} \\ \begin{pmatrix} \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \end{pmatrix}\langle\theta\rangle &= \begin{matrix} \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c_1\langle\theta\rangle \textbf{ else } c_2\langle\theta\rangle \end{matrix} \\ (c_1 \textbf{ and } c_2)\langle\theta\rangle &= c_1\langle\theta\rangle \textbf{ and } c_2\langle\theta\rangle \\ (c_1 \textbf{ or } c_2)\langle\theta\rangle &= c_1\langle\theta\rangle \textbf{ or } c_2\langle\theta\rangle \\ (\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2)\langle\theta\rangle &= \textbf{if } e \textbf{ then } c_1\langle\theta\rangle \textbf{ else } c_2\langle\theta\rangle \\ f(\vec{e})\langle e'_1, \dots, e'_n\rangle\langle\theta\rangle &= f(\vec{e})\langle e'_1[\theta], \dots, e'_n[\theta]\rangle \end{aligned}$$

Fig. 7. Substitution of value parameters into expressions $e[\theta]$, deadline expressions $d[\theta]$, and clauses $c[\theta]$; and substitution of party parameters into clauses $c\langle\theta\rangle$.

In Fig. 7, we define two types of applications of substitutions to CSL constructs: substitutions of value parameters in (deadline) expressions and clauses, denoted $e[\theta]$, $d[\theta]$, and $c[\theta]$, respectively, where θ is a substitution; and substitution of party parameters in clauses, denoted $c\langle\theta\rangle$, where θ is a party substitution. We write $c[v/x]$ for the application on clause c of the substitution that maps x to v. Also, $c[\vec{v}/\vec{x}] = c[v_1/x_1]\dots[v_n/x_n]$ for vectors $\vec{v} = (v_1, \dots, v_n)$ and $\vec{x} = (x_1, \dots, x_n)$. Finally, we abuse notation by interpreting vectors of variables as sets in Fig. 7.

The following lemma shows that the substitutions defined in Fig. 7 fulfil the expected properties with respect to the type system. Moreover, party parameters are typed using *relevance typing* [23], that is parametrised parties are used at least once in the body of a template definition.

Lemma 9. *Consider a well-typed expression $\Gamma \vdash e : t$, a well-typed deadline expression $\Gamma \vdash d : Deadline$, and a well-typed clause $\Delta, \Lambda, \Gamma \vdash c : Clause\langle P\rangle$. For any substitution θ such that $\theta(x) \in [\![\Gamma(x)]\!]$ for all $x \in \mathrm{dom}(\theta) \cap \mathrm{dom}(\Gamma)$, it holds that*

$$\begin{aligned} \Gamma' &\vdash e[\theta] : t, \\ \Gamma' &\vdash d[\theta] : Deadline, \\ \Delta, \Lambda, \Gamma' &\vdash c[\theta] : Clause\langle P\rangle, \end{aligned}$$

where $\Gamma' = \Gamma|_{\mathrm{dom}(\Gamma)\setminus\mathrm{dom}(\theta)}$. Moreover, for any party substitution θ, it holds that

$$\Delta, \Lambda\setminus\mathrm{dom}(\theta), \Gamma \vdash c\langle\theta\rangle : Clause\langle P \cup \{p \mid \theta(x) = p, x \in \mathrm{dom}(\Lambda) \cap \mathrm{dom}(\theta)\}\rangle.$$

Proof. The first typing judgment (that is $\Gamma' \vdash e[\theta] : t$) follows easily by induction on the typing derivation $\Gamma \vdash e : t$, and the second judgment then follows immediately. The third judgment follows by induction on the typing derivation $\Delta, \Lambda, \Gamma \vdash c : \mathrm{Clause}\langle P\rangle$, and the same goes for the fourth judgment. $\square$

$$\boxed{D,\tau \vdash c \xrightarrow{\epsilon} \mathbf{c}} \qquad \frac{}{D,\tau \vdash \textbf{fulfilment} \xrightarrow{\epsilon} \textbf{fulfilment}}$$

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1,\tau_2) \qquad \tau_1 \le \tau' \le \tau_2}{D,\tau \vdash \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau',k(\vec{v}))} c[\vec{v}/\vec{x},\tau_2-\tau'/z]}$$

$$\frac{d \Downarrow^{\tau} (\tau_1,\tau_2) \qquad \tau' > \tau_2}{D,\tau \vdash \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau',k'(\vec{v}))} (\max(\tau,\tau_2),\{p\})}$$

$$\frac{d \Downarrow^{\tau} (\tau_1,\tau_2) \quad \tau' \le \tau_2 \qquad \tau' < \tau_1 \text{ or } k' \ne k \text{ or } e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \qquad d' = \textbf{after}\ \tau_1-\tau'\ \textbf{within}\ \tau_2-\tau_1}{\begin{array}{l} D,\tau \vdash \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau',k'(\vec{v}))} \\ \qquad\quad \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d'\ \textbf{remaining}\ z\ \textbf{then}\ c \end{array}}$$

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1,\tau_2) \qquad \tau_1 \le \tau' \le \tau_2}{D,\tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau',k(\vec{v}))} c_1[\vec{v}/\vec{x},\tau_2-\tau'/z]}$$

$$\frac{d \Downarrow^{\tau} (\tau_1,\tau_2) \qquad \tau' > \tau_2 \qquad D,\max(\tau,\tau_2) \vdash c_2 \xrightarrow{(\tau',k'(\vec{v}))} \mathbf{c}}{D,\tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau',k'(\vec{v}))} \mathbf{c}}$$

$$\frac{d \Downarrow^{\tau} (\tau_1,\tau_2) \quad \tau' \le \tau_2 \qquad \tau' < \tau_1 \text{ or } k' \ne k \text{ or } e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \qquad d' = \textbf{after}\ \tau_1-\tau'\ \textbf{within}\ \tau_2-\tau_1}{\begin{array}{l} D,\tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau',k'(\vec{v}))} \\ \qquad\quad \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d'\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \end{array}}$$

$$\frac{D,\tau \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1 \qquad D,\tau \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2}{D,\tau \vdash c_1\ \textbf{and}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_1 \varowedge \mathbf{c}_2} \qquad \frac{D,\tau \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1 \qquad D,\tau \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2}{D,\tau \vdash c_1\ \textbf{or}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_1 \varovee \mathbf{c}_2}$$

$$\frac{e \Downarrow \textbf{true} \qquad D,\tau \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1}{D,\tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_1} \qquad \frac{e \Downarrow \textbf{false} \qquad D,\tau \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2}{D,\tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_2}$$

$$\frac{\vec{e} \Downarrow \vec{v} \qquad (f(\vec{x})\langle\vec{y}\rangle = c) \in D \qquad D,\tau \vdash c[\vec{v}/\vec{x},\vec{p}/\vec{y}]\langle\vec{p}/\vec{y}\rangle \xrightarrow{\epsilon} \mathbf{c}}{D,\tau \vdash f(\vec{e})\langle\vec{p}\rangle \xrightarrow{\epsilon} \mathbf{c}}$$

$$\boxed{s \xrightarrow{\epsilon} \mathbf{s}} \qquad \frac{D,\tau \vdash c \xrightarrow{\epsilon} (\tau',B)}{\textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau \xrightarrow{\epsilon} (\tau',B)}$$

$$\frac{D,\tau \vdash c \xrightarrow{\epsilon} c' \qquad \mathrm{ts}(\epsilon) = \tau'}{\textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau \xrightarrow{\epsilon} \textbf{letrec}\ D\ \textbf{in}\ c'\ \textbf{starting}\ \tau'}$$

Fig. 8. Reduction semantics for CSL clauses *c* and specifications *s*.

The reduction semantics for well-formed specifications is presented in Fig. 8. The reduction relation for clauses has the form $D, \tau \vdash c \xrightarrow{\epsilon} \mathbf{c}$, where D is a set of template definitions, τ is the time of the last update to the contract (initially the starting time), c is the clause to reduce, ϵ is the event that takes place, and $\mathbf{c}$ is the *residue*. A residue $\mathbf{c}$ is either a clause, representing the remaining obligations, or a breach of contract.

The second, third, and fourth rules describe the three different situations for obligations: (1) either the event fulfils the obligation, and the residue is determined by the continuation clause; or (2) the event does not fulfil the obligation by missing the deadline, in which case a breach of contract takes place; or (3) the event does not fulfil the obligation, but nor does it violate the deadline, so the obligation—with updated deadlines—remains the residue. The three rules for external choice are similar, except that in the second case the residue is determined by the alternative branch of the choice, rather than a breach of contract.

It follows from the operational semantics that a clause can only be breached by missing a deadline, and the time of breach is determined by the deadline itself. However, we need to take into account that deadlines may be negative, in which case we define the time of breach as the time of the last update. Similarly, we need to take negative deadlines into account for external choices. Note that in the rules, clauses are fully instantiated, that is they have no free variables (for the straightforward definition of free variables): the type system guarantees that well-typed clauses are fully instantiated, as we shall see shortly.

The semantics of clause conjunction and clause disjunction use lifted versions of the corresponding verdict compositions, which are defined by:

$$\mathbf{c}_1 \owedge \mathbf{c}_2 = \begin{cases} c_1 \textbf{ and } c_2 & \text{if } \mathbf{c}_1 = c_1 \text{ and } \mathbf{c}_2 = c_2, \\ (\tau_1, B_1) & \text{if } \mathbf{c}_1 = (\tau_1, B_1) \text{ and } \mathbf{c}_2 = c_2, \\ (\tau_2, B_2) & \text{if } \mathbf{c}_2 = (\tau_2, B_2) \text{ and } \mathbf{c}_1 = c_1, \\ (\tau_1, B_1) \wedge (\tau_2, B_2) & \text{if } \mathbf{c}_1 = (\tau_1, B_1) \text{ and } \mathbf{c}_2 = (\tau_2, B_2) \end{cases}$$

and

$$\mathbf{c}_1 \ovee \mathbf{c}_2 = \begin{cases} c_1 \textbf{ or } c_2 & \text{if } \mathbf{c}_1 = c_1 \text{ and } \mathbf{c}_2 = c_2, \\ c_1 & \text{if } \mathbf{c}_1 = c_1 \text{ and } \mathbf{c}_2 = (\tau_2, B_2), \\ c_2 & \text{if } \mathbf{c}_2 = c_2 \text{ and } \mathbf{c}_1 = (\tau_1, B_1), \\ (\tau_1, B_1) \vee (\tau_2, B_2) & \text{if } \mathbf{c}_1 = (\tau_1, B_1) \text{ and } \mathbf{c}_2 = (\tau_2, B_2). \end{cases}$$

The reduction semantics is lifted to specifications $s \xrightarrow{\epsilon} \mathbf{s}$, where the residue $\mathbf{s}$ is either a residual specification or a breach of contract. Note that the time of the last update (that is event) is recorded in the residual specification.

The following theorem shows that the semantics satisfies *type preservation* [24]. Moreover, the set of parties in the typing of the residual specification may decrease, matching the intuition that parties may become free of obligations during the execution of a contract.

Theorem 10. *Let s be a well-formed specification involving parties P and s' be a specification. If $s \xrightarrow{\epsilon} s'$ then s' is a well-formed specification involving parties P', for some $P' \subseteq P$.*

Proof. The proof is deferred to page 93. The proof is by induction on the typing derivation. □

The following theorem shows that the semantics also satisfies the *progress property* [24], that is well-formed specifications never get stuck.

Theorem 11. *Let s be a well-formed specification with parties P and starting time τ_0. Then for any event ϵ with $\text{ts}(\epsilon) \geq \tau_0$ there is a unique residue $\mathbf{s}$ such that $s \xrightarrow{\epsilon} \mathbf{s}$. Furthermore, whenever $\mathbf{s} = (\tau, B)$ then $\tau_0 \leq \tau \leq \text{ts}(\epsilon)$ and $B \subseteq P$.*

Proof. The proof is deferred to page 96. The proof is by nested induction on the structure of the immediate unfolding relation and the step derivation. □

3.5. Mapping CSL specifications to contracts

The reduction semantics presented in Section 3.4 is event-driven: at the occurrence of an event, a specification reduces to either a breach of contract or a residual specification. However, the absence of events is also significant, because it may imply that the contract execution is considered finished and no more events are produced. In this case a verdict needs to be associated with the residual specification. Formally, we associate the verdict ν with a specification s if $\vdash s \downarrow \nu$ can be derived using the derivation rules of Fig. 9. For any well-formed specification s, there exists a unique verdict ν associated with s.

We can now associate a verdict with a specification and an event trace by running the specification on the trace: at each step the specification is reduced on the current event, until either a breach occurs or there are no more events, in which case we check if the residual specification is fulfilled according to the relation in Fig. 9. Formally, the function $[\![s]\!] : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ where τ_0 is the start time of s, is defined on finite traces inductively by:

$$[\![s]\!](\sigma) = \begin{cases} \nu & \text{if } \sigma = \langle\rangle \text{ and } \vdash s \downarrow \nu, \\ (\tau, B) & \text{if } \sigma = \epsilon\sigma' \text{ and } s \xrightarrow{\epsilon} (\tau, B), \\ [\![s']\!](\sigma') & \text{if } \sigma = \epsilon\sigma' \text{ and } s \xrightarrow{\epsilon} s', \end{cases}$$

and on infinite traces by the (unique) extension in Lemma 3.

$$\boxed{D, \tau \vdash c \downarrow \nu} \qquad \frac{}{D, \tau \vdash \textbf{fulfilment} \downarrow \checkmark}$$

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2)}{D, \tau \vdash \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \downarrow (\max(\tau, \tau_2), \{p\})}$$

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad D, \max(\tau, \tau_2) \vdash c_2 \downarrow \nu_2}{D, \tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \downarrow \nu_2}$$

$$\frac{e \Downarrow \textbf{true} \qquad D, \tau \vdash c_1 \downarrow \nu_1}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \downarrow \nu_1} \qquad \frac{e \Downarrow \textbf{false} \qquad D, \tau \vdash c_2 \downarrow \nu_2}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \downarrow \nu_2}$$

$$\frac{D, \tau \vdash c_1 \downarrow \nu_1 \qquad D, \tau \vdash c_2 \downarrow \nu_2}{D, \tau \vdash c_1\ \textbf{and}\ c_2 \downarrow \nu_1 \wedge \nu_2} \qquad \frac{D, \tau \vdash c_1 \downarrow \nu_1 \qquad D, \tau \vdash c_2 \downarrow \nu_2}{D, \tau \vdash c_1\ \textbf{or}\ c_2 \downarrow \nu_1 \vee \nu_2}$$

$$\frac{\vec{e} \Downarrow \vec{v} \qquad f(\vec{x})\langle \vec{y} \rangle = c \in D \qquad D, \tau \vdash c[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle \vec{p}/\vec{y} \rangle \downarrow \nu}{D, \tau \vdash f(\vec{e})\langle \vec{p} \rangle \downarrow \nu}$$

$$\boxed{\vdash s \downarrow \nu} \qquad \frac{D, \tau \vdash c \downarrow \nu}{\vdash \textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau \downarrow \nu}$$

Fig. 9. Verdict ν associated with specification s.

The following theorem shows that CSL specifications indeed represent contracts in the sense of Definition 1.

Theorem 12. *Let s be a well-formed specification with parties P and start time τ_0. Then $[\![s]\!]$ is a contract between parties P starting at time τ_0.*

Proof. The proof is deferred to page 97. The proof follows by induction on the length of the trace using Theorems 10 and 11. $\square$

Corollary 13. *Let $s =$ **letrec** D **in** c **starting** τ be a well-formed specification. Then*

$$[\![s]\!] = \begin{cases} \mathsf{c}_{\checkmark} & \text{if } c = \textbf{fulfilment}, \\ [\![s_1]\!] \wedge [\![s_2]\!] & \text{if } c = c_1\ \textbf{and}\ c_2, \\ [\![s_1]\!] \vee [\![s_2]\!] & \text{if } c = c_1\ \textbf{or}\ c_2, \end{cases}$$

*where $s_i =$ **letrec** D **in** c_i **starting** τ.*

Proof. For finite traces the proofs follow by induction on the trace length, similar to the proof of Theorem 12. For infinite traces the results then follow from the uniqueness result of Lemma 3. $\square$

The theorem and its corollary show that CSL enjoys the principles underpinning the contract model defined in Section 2, that is deterministic blame assignment and compositionality. Moreover, the algebraic properties stated in Corollary 7 carry over to CSL.

3.6. Monitoring CSL specifications

The reduction semantics presented above gives rise to an incremental run-time monitoring algorithm for CSL specifications. The main ingredient of the monitor is the function $\text{mon} : \mathsf{S} \times \mathsf{Tr}^{\tau_0}_{\text{fin}} \to (\mathsf{V}_! \cup \mathsf{V}_?) \times \mathsf{S}$ defined by

$$\text{mon}(s, \sigma) = \begin{cases} (\nu_?, s) & \text{if } \sigma = \langle\rangle \text{ and } \vdash s \downarrow \nu, \\ (\nu_!, s') & \text{if } \sigma = \sigma'\epsilon \text{ and } \text{mon}(s, \sigma') = (\nu_!, s'), \\ ((\tau, B)_!, s') & \text{if } \sigma = \sigma'\epsilon \text{ and } \text{mon}(s, \sigma') = (\nu_?, s') \text{ and } s' \xrightarrow{\epsilon} (\tau, B), \\ (\nu_?, s'') & \text{if } \sigma = \sigma'\epsilon \text{ and } \text{mon}(s, \sigma') = (\nu'_?, s') \text{ and } s' \xrightarrow{\epsilon} s'' \text{ and } \vdash s'' \downarrow \nu, \end{cases}$$

where S is the set of all well-formed CSL specifications.

The monitor is invoked whenever an event occurs, provided that the monitor has not already output a final verdict. Between invocations, it only needs to remember the previous result, that is in order to process the event ϵ, after the events σ have happened, we only need the previous result $\text{mon}(s, \sigma)$ in order to compute the new result $\text{mon}(s, \sigma\epsilon)$.

The function mon is not a run-time monitor in the sense of Definition 6. However, it is very close to one, as shown by the following theorem, which follows directly from Theorem 12.

Theorem 14. *Let s be a specification with starting time τ_0. The function* mon *is computable and for any trace $\sigma \in \mathrm{Tr}_{\mathrm{fin}}^{\tau_0}$ verdict $\nu_\star$, and residual specification s', with $\mathrm{mon}(s, \sigma) = (\nu_\star, s')$, it holds that*

(1) if $\nu_\star = (\tau, B)_!$ then $[\![s]\!](\sigma') = (\tau, B)$ for all σ' with $\sigma \sqsubset \sigma'$,
(2) if $\nu_\star = \checkmark_?$ then $[\![s]\!](\sigma) = \checkmark$, and
(3) if $\nu_\star = (\tau, B)_?$ then $[\![s]\!](\sigma) = (\tau, B)$ and $\tau \geq \mathrm{end}(\sigma)$.

The result above shows that our run-time monitor satisfies impartiality (1), however it does not always satisfy anticipation. For instance, if the body of a specification is **fulfilment**, then our monitor always outputs $\checkmark_?$, even if anticipation requires that it outputs $\checkmark_!$. Building a run-time monitor which guarantees anticipation is hard, because the expression language can "hide" anticipated verdicts. Consider for instance the clauses

$$c_1 = \langle p\rangle\ k(x)\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c,$$
$$c_2 = \textbf{if}\ k(x)\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c\ \textbf{else fulfilment},$$

where e is some expression for which $e[v/x] \Downarrow$ **false** for all values v, for instance $e = x > 0 \wedge x < 0$. The contract represented by c_1 is always breached, while the one represented by c_2 is never breached. Hence, in order to guarantee anticipation, one first needs to decide satisfiability for the expression language.

Example 6. We demonstrate the reduction semantics and run-time monitor using the CSL specification in Fig. 3. As in Example 4, we consider the trace $\langle \epsilon_1, \epsilon_3, \epsilon_4\rangle$, where the events are as in the example, except that they use concrete actions instead of abstract actions:

$$\epsilon_1 = (\text{2011-01-01}, \text{Deliver(Seller, Buyer, “Laser printer”)}),$$
$$\epsilon_3 = (\text{2011-01-01}, \text{Payment(Buyer, Seller, 100)}),$$
$$\epsilon_4 = (\text{2011-01-10}, \text{Payment(Buyer, Seller, 100)}).$$

We first define the specifications s_i, with $i \in \{0, 1, 2, 3\}$:

$s_i =$ **letrec** *sale*(*deliveryDeadline*, *goods*, *payment*)⟨*buyer*, *seller*⟩ = *c* **in** $c_i[\theta]$ **starting** 2011-01-01

where θ(*deliveryDeadline*) = 0, θ(*goods*) = "Laser printer", θ(*payment*) = 200, θ(*buyer*) = Buyer, θ(*seller*) = Seller, and

```
c₀ = sale(0, “Laser printer”, 200)⟨Buyer, Seller⟩
c = ⟨seller⟩ Deliver(s,r,g) where s = seller ∧ r = buyer ∧ g = goods due within deliveryDeadline then c₁
c₁ = ⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = payment/2 due immediately then c₂
c₂ = (⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = payment/2 due within 30D
      or
      ⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = (payment/2) ∗ 110/100 due within 14D after 30D)
     and
     if Return(s,r,g) where s = buyer ∧ r = seller ∧ g = goods due within 14D
     then
     ⟨seller⟩ Payment(s,r,a) where s = seller ∧ r = buyer ∧ a = payment due within 7D
c₃ = if Return(s,r,g) where s = buyer ∧ r = seller ∧ g = goods due after −9D within 5D
     then
    ⟨seller⟩ Payment(s,r,a) where s = seller ∧ r = buyer ∧ a = payment due within 7D
```

The specification in Fig. 3 equals s_0. We have $s_0 \xrightarrow{\epsilon_1} s_1 \xrightarrow{\epsilon_3} s_2 \xrightarrow{\epsilon_4} s_3$. Note that the relative deadline in c_3 for returning the goods is shifted with regard to the corresponding relative deadline in c_2, due to the passing of time. The incremental output of the monitor on the trace $\langle \epsilon_1, \epsilon_3, \epsilon_4\rangle$ is as follows:

$$\begin{aligned}
\mathrm{mon}(s_0, \langle\rangle) &= ((\text{2011-01-01}, \{\text{Seller}\})_?, s_0),\\
\mathrm{mon}(s_0, \langle\epsilon_1\rangle) &= ((\text{2011-01-01}, \{\text{Buyer}\})_?, s_1),\\
\mathrm{mon}(s_0, \langle\epsilon_1, \epsilon_3\rangle) &= ((\text{2011-02-14}, \{\text{Buyer}\})_?, s_2),\\
\mathrm{mon}(s_0, \langle\epsilon_1, \epsilon_3, \epsilon_4\rangle) &= (\checkmark_?, s_3).
\end{aligned}$$

Finally, remark that on all traces, except the last one, the value of mon coincides with the value of the run-time monitor of Definition 4.

Paragraph 1. The following agreement is enacted on 2011-01-01, and is valid for 5 years.

Paragraph 2. The Employee agrees not to disclose any information regarding the work carried out under the Employer, as stipulated in Paragraph 3.

Paragraph 3. (Omitted.)

— ◇ —

```
letrec nda()⟨employee⟩ =
 if Disclose(e) where e = employee due within 5 Y then
   ⟨employee⟩ Unfulfillable where false due immediately
in
nda()⟨Employee⟩ starting 2011-01-01
```

Fig. 10. A non-disclosure agreement (paper version top, CSL version bottom).

3.7. Contract examples

We have seen one example of a realistic contract specified in CSL, namely the sales contract in Fig. 1. The example illustrates how dependencies between paragraphs are realised as continuation clauses, how obligations and permissions are represented, and how contract disjunction enables choices. In this section we provide further specification examples, which illustrate prohibitions, potentially infinite contracts, linear treatment of events (as in linear logic [6]), and a more involved application of arithmetic expressions.

Prohibitions: Prohibitions are not built-in to CSL, yet it is possible to express prohibitions using external choices and obligations. Consider the *non-disclosure agreement* in Fig. 10 (top). The agreement is formalised in Fig. 10 (bottom), using a signature that includes the action kinds {Disclose, Unfulfillable} $\subseteq \mathcal{K}$, with types ar(Disclose) = ⟨Party⟩ and ar(Unfulfillable) = ⟨⟩. We use the action kind Unfulfillable to point out that the corresponding obligation cannot be fulfilled. Besides the technique for encoding prohibitions, the example illustrates an important point, namely that we do not model how parties agree that events have taken place. In the agreement above, a dispute is more likely to involve proving (or disproving) disclosure of information, rather than interpreting whether disclosing information is allowed or not.

Lease agreement: The next example is a lease agreement presented in Fig. 11 (top). The contract is formalised in Fig. 11 (bottom), using a signature that includes the action kinds {Payment, ReqTermination, Provide} $\subseteq \mathcal{K}$, with types ar(Payment) = ⟨Party, Party, Int⟩, ar(ReqTermination) = ⟨Party⟩, and ar(Provide) = ⟨Party, Party, String, Int⟩. We assume that the expression language has been extended with a function for calculating the minimum of two integers.

The example demonstrates how recursive template definitions enable potentially infinite contracts: each lease period is guaranteed to be executed at least 6 times, but there is no a priori upper bound on the number of iterations. The example also illustrates the usage of the **remaining** construct, which is needed in order to determine the start of the next lease period, when one of the parties requests a termination.

Master sales agreement: Next we consider a master sales agreement in Fig. 12 (top). The contract is formalised in Fig. 12 (bottom), using a signature that includes the action kinds {Request, IssueInvoice, Deliver, Payment} $\subseteq \mathcal{K}$, with types ar(Request) = ⟨Party, Party, Int, String⟩, ar(IssueInvoice) = ⟨Party, Party, Int⟩, ar(Deliver) = ⟨Party, Party, Int, String, Int⟩, and ar(Payment) = ⟨Party, Party, Int, Int⟩. We assume that the expression language has been extended with a function for calculating the maximum of two integers.

The encoding illustrates the usage of multiple template definitions and that deadlines can be calculated dynamically based on previous events. Moreover, the action kinds pertaining to each individual sale contain identifiers that are needed in order to distinguish potentially identical payments, deliveries, or invoices when there are simultaneous orders.

Instalment sale: The last contract we consider is an instalment sale in Fig. 13 (top). For simplicity, we have only included the payment part of the contract, and not sellers obligation to deliver goods. The CSL formalisation is presented in Fig. 13 (bottom), and it shows a more involved application of in-place arithmetic expressions, namely calculation of the remaining balance after each instalment has been payed. Note that contract termination not only depends on the initial 24 months period, but that the contract may end earlier, in case the remaining balance is fully payed.

4. Related work

Formal specification of contracts and automatic reasoning about contracts has drawn interest from a wide variety of research areas within computer science, going back to the late eighties with the pioneering work by Lee [12]. Contract formalisms typically fall into three categories: (deontic) logic based formalisms [9,12,26], event-condition-action based formalisms [7,14], and trace based formalisms [2,11]. The logic based approaches mainly focus on declarative specification of contracts, and on (meta) reasoning, such as decidability of the logic. On the other hand, the event-condition-action and trace based models focus mainly on contract execution. The latter approach takes a more extensional view of contracts, that is contracts are denoted by the set of traces they accept. Other approaches to contract modelling include combinator

Paragraph 1. The term of this lease is for 6 months, beginning on 2011-01-01. At the expiration of said term, the lease will automatically be renewed for a period of one month unless either party (Landlord or Tenant) notifies the other of its intention to terminate the lease at least one month before its expiration date.

Paragraph 2. The lease is for 1 apartment, which is provided by Landlord throughout the term.

Paragraph 3. Tenant agrees to pay the amount of €1000 per month, each payment due on the 7th day of each month.

— ◇ —

```
letrec lease(property, leaseStart, leasePeriod, leasePeriods, payment, payDeadline,
             terminationRequested)⟨lessor, lessee⟩ =
 if leasePeriods ≤ 0 ∧ terminationRequested then
   fulfilment
 else
   ⟨lessee⟩ Payment(s,r,a) where s = lessee ∧ r = lessor ∧ a = payment
                           due immediately after leaseStart + payDeadline
   and
   ⟨lessor⟩ Provide(s,r,p,l) where s = lessor ∧ r = lessee ∧ p = property ∧ l = leasePeriod
                           due immediately after leaseStart
   then if terminationRequested then
     lease(property, leasePeriod, leasePeriod, leasePeriods − 1, payment,
           payDeadline, true)⟨lessor, lessee⟩
   else if ReqTermination(s) where s = lessor ∨ s = lessee
                              due within leasePeriod remaining z
   then
     lease(property, z, leasePeriod, min(1,leasePeriods − 1), payment,
           payDeadline, true)⟨lessor, lessee⟩
   else
     lease(property, 0, leasePeriod, leasePeriods − 1, payment,
           payDeadline, false)⟨lessor, lessee⟩
in
lease("Apartment", 0, 1M, 6, 1000, 7D, false)⟨Landlord, Tenant⟩ starting 2011-01-01
```

Fig. 11. A lease agreement (paper version top, CSL version bottom).

libraries [22], defeasible reasoning [8,10,27], commitment graphs, that is graph theoretic representations of responsibility between parties [31,32], finite state machines [17], and more informal frameworks [3,4,18,29]. Common to all approaches is the goal of modelling (electronic) contracts in general, except for Peyton-Jones and Eber [22], Andersen et al. [2], and Tan and Thoen [27] who specifically consider financial contracts, commercial contracts, and trade contracts, respectively.

Existing contract frameworks tend to focus either on contract execution models [17,18,31,32], or on concrete specification languages [2–4,7,9,10,12,22,26,27], rather than considering both an abstract semantic model and a specification language. Consequently, these frameworks either lack a language for specifying contracts, or they lack an operational interpretation—with the exception of [2,26], who however do not characterise contracts abstractly in terms of their semantic models. In contrast, we consider both an abstract execution model and a specification language. Besides giving a formal operational interpretation to specifications, this makes it possible to consider different specification languages for different contract domains, and still compare their semantics in terms of the abstract model. Moreover, by mapping a specification language into our model, deterministic blame assignment is guaranteed, algebraic properties of conjunction and disjunction follow automatically, and run-time monitoring has a well-defined meaning.

Compared with the previous contract execution models [17,18,31,32], our abstract contract model relies on fewer high-level concepts. For instance, the existing models rely on concepts such as deadlines [31,32], deontic modalities [17] and logical formulae [18], which are all definable within our model.

Compared with the previous contract specification languages [2–4,7,9,10,12,22,26,27], ours mainly distinguishes itself by incorporating deterministic blame assignment. Besides, existing languages all fall short of other important features. History sensitive commitments, that is commitments which depend on what has happened in the past, are only supported in few languages [2,9]. History sensitivity is typically not supported because actions are modelled as propositional variables, hence actions cannot carry values. Only the language of Andersen et al. [2] has support for (recursive) contract templates; we have adapted their construction to CSL. Furthermore, potentially infinite contracts are only supported in few languages [2,12,26]. Finally, some languages lack absolute temporal constraints [8,10,26], and instead consider only relative temporal constraints.

The importance of monitoring contracts is widely recognised [2,7,9,17,26,31,32], yet few authors provide a formal, operational semantics for contract execution [2,26]. Such a semantics is a prerequisite for showing that a monitor achieves its goals. Furthermore, deterministic blame assignment is crucial for run-time monitoring, a feature which—to the best of our knowledge—has only previously been recognised by Xu and Jeusfeld [32]. However, Xu and Jeusfeld only consider

Paragraph 1. The master agreement between Vendor and Customer is for 1000 printers, with a unit price of €100. The agreement is valid for one year, starting 2011-01-01.

Paragraph 2. The customer may at any time order an amount of printers (with the total not exceeding the threshold of 1000), after which the Vendor must deliver the goods before the maximum of (i) 14 days, or (ii) the number of ordered goods divided by ten days.

Paragraph 3. After delivering the goods, Vendor may invoice the Customer within 1 month, after which the goods must be paid for by Customer within 14 days.

— ◇ —

```
letrec master(goods, amount, terminationDeadline, payment, invoiceDeadline,
              paymentDeadline, id)⟨vendor, customer⟩ =
 if amount = 0 then
   fulfilment
 else if Request(s,r,n,g) where s = customer ∧ r = vendor ∧ n ≤ amount ∧
                                n > 0 ∧ g = goods
                          due within terminationDeadline remaining z
 then
   sale(n, g, n*payment, max(14D,n*24*60*6),
        invoiceDeadline, paymentDeadline, id)⟨vendor, customer⟩
   and
   master(goods, amount − n, z, payment,
          invoiceDeadline, paymentDeadline, id + 1)⟨vendor, customer⟩

sale(number, goods, payment, deliveryDeadline, invoiceDeadline, paymentDeadline, id)
    ⟨seller, buyer⟩ =
 ⟨seller⟩ Deliver(s,r,n,g,i)
        where s = seller ∧ r = buyer ∧ n = number ∧ g = goods ∧ i = id
        due within deliveryDeadline
 then
 if IssueInvoice(s,r,i) where s = seller ∧ r = buyer ∧ i = id
                        due within invoiceDeadline
 then
   ⟨buyer⟩ Payment(s,r,a,i) where s = buyer ∧ r = seller ∧ a = payment ∧ i = id
                            due within paymentDeadline
in
master("Printer", 1000, 1Y, 100, 1M, 14D, 0)⟨Vendor, Customer⟩ starting 2011-01-01
```

Fig. 12. Master sales agreement (paper version top, CSL version bottom).

monitoring and blame assignment for their particular specification language, while we also define these notions in a general and abstract setting.

Compositional specification of contracts is traditionally obtained by means of conjunction and disjunction [2,9,22,26]. Besides, Andersen et al. [2] present a language which supports linear conjunction [6]. Despite the fact that compositionality of contracts has previously been considered, there has been no previous treatment of the effect of compositionality on blame assignment, and in particular on how disjunctions involving different parties may give rise to nondeterminism.

Standard deontic logic (SDL) [28]—the logic of obligations, permissions, and prohibitions—has inspired existing contract formalisms [9,12,26] due to the appealing similarities with concepts from contracts. Yet the *possible worlds* semantics [30] of deontic logic lacks an operational interpretation, which in our view makes SDL inappropriate as a basis for formalising contracts. To alleviate this weakness, Prisacariu and Schneider [26] consider a restricted form of deontic modalities with *ought-to-do* rather than *ought-to-be*, meaning that deontic modalities are only to specify what should happen ("Seller ought to deliver"), and not what should be the general state of affairs ("It ought to be the case that Seller delivers"). The restriction to ought-to-do statements gives rise to an alternative μ-calculus semantics based on actions. We also restrict contracts to ought-to-do statements.

It has been argued that contrary-to-duty obligations [25]—also a SDL related concept—are crucial for contracts as well [3,9,18,26]. Although we recognise the importance of reparation activities in contracts, we instead consider them ordinary choices, rather than choices with an implicit agreement to conform first and foremost with primary objectives. In consideration hereof, we avoid the philosophical considerations of contrary-to-duty [9,25], and the treatment of intermediate violations generated by failing to comply with primary objectives.

5. Conclusions

In this article we have presented a novel, trace-based model for multiparty contracts with blame assignment. We have illustrated that high-level contract concepts such as obligations, deadlines, and reparation clauses are representable within our model. This shows that our model is well-suited for representing real-world contracts. For the purpose of writing

Paragraph 1. Buyer agrees to pay to Seller the total sum €10000, in the manner following:

Paragraph 2. €500 is to be paid at closing, and the remaining balance of €9500 shall be paid as follows:

Paragraph 3. €500 or more per month on the first day of each and every month, and continuing until the entire balance, including both principal and interest, shall be paid in full; provided, however, that the entire balance due plus accrued interest and any other amounts due hereunder shall be paid in full on or before 24 months.

Paragraph 4. Monthly payments shall include both principle and interest with interest at the rate of 10%, computed monthly on the remaining balance from time to time unpaid.

— ◇ —

```
letrec instalments(balance, instalment, payDeadline, start, end, frequency,
                   rate, closingPayment, seller)⟨buyer⟩ =
 if balance ≤ 0 then
   ⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = closingPayment
                          due within end
 else if end ≤ start then
   ⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a = balance + closingPayment
                          due within end
 else
   ⟨buyer⟩ Payment(s,r,a) where s = buyer ∧ r = seller ∧ a ≥ min(balance,instalment) ∧
                                a ≤ balance
                          due within payDeadline after start remaining z
   then
   instalments (((100 + rate) ∗ (balance − a)) / 100, instalment,
                payDeadline, frequency − payDeadline + z,
                end − start − payDeadline + z,
                frequency, rate, closingPayment, seller)⟨buyer⟩
in
instalments(10000, 500, 1D, 0, 24M, 1M, 10, 500, Seller)⟨Buyer⟩ starting 2011-01-01
```

Fig. 13. Instalment sale (paper version top, CSL version bottom).

contracts, we have given a contract specification language, which enjoys the principle of blame assignment by inheritance from the abstract model, and which is amenable to incremental run-time monitoring.

We plan to use CSL in case studies to further evaluate its applicability for formalising contracts and monitoring their executions. Here, we expect that the expression language of CSL needs to be extended, while hopefully the clause language does not require additions. The extensions to the expression language should be straightforward.

A restriction in our model is that blame is deterministically assigned to contract parties in case of breach of contract. Although deterministic blame assignment is a desired feature, not all real-world contracts have this feature. In future work, we plan to extend our model such that verdicts can be nondeterministically associated with traces. Such an extension is also motivated by the objective for obtaining less restrictive operators for composing contracts.

Future work also includes contract analysis. Such an analysis can be based on our abstract contract model or on the reduction semantics of CSL. For instance, an immediately implementable online analysis based on the reduction semantics is to simulate the outcome of possible future events. Together with the information on who is responsible for an event, this is useful to avoid a breach of contract and to issue reminders of deadlines. The monitoring algorithm partly does this already by outputting potential breaches which represent upcoming deadlines. A further goal of such an online analysis is to monitor contract execution with full anticipation. However, in order to effectively perform such monitoring of CSL specifications, it may be necessary to restrict oneself to fragments of CSL. Other contract analyses are (1) satisfiability, that is whether a contract can be fulfilled at all, (2) satisfiability with respect to a particular party, that is whether a party can avoid breaching a contract in which it is involved, (3) contract valuation, that is what is the expected value of a contract for a given party, and (4) contract entailment, that is whether fulfilling a contract entails the fulfilment of another contract. The last analysis has applications for instance in checking contract conformance with regulations, when regulations are themselves formalised as contracts.

Acknowledgements

The authors thank the participants of the FLACOS 2010 workshop for providing valuable input to an early state of this work. The first author would also like to thank David Basin for the invitation to visit his research group at ETH Zurich in the first half of 2010, during which foundations of the work presented in this article were laid.

Appendix A. Additional proof details

Proof of Theorem 10. Assume that s is well-formed with parties P, that is $\vdash s : \mathrm{Contract}\langle P\rangle$ and the unfolding relation on the template names of s is acyclic. Assume furthermore that $s \xrightarrow{\epsilon} s'$, where $s =$ **letrec** D **in** c **starting** τ. Then $s' =$

letrec D **in** c' **starting** $\mathrm{ts}(\epsilon)$, for some clause c', where $D, \tau \vdash c \xrightarrow{\epsilon} c'$. We need to show that s' is well-formed with parties $P' \subseteq P$, which amounts to showing that $\vdash s' : \mathrm{Contract}\langle P'\rangle$ as the templates of s and s' are identical. Since s is well-typed, we have that $\Delta \vdash D$ and $\Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P\rangle$, so it suffices to show that $\Delta, \emptyset, \emptyset \vdash c' : \mathrm{Clause}\langle P'\rangle$ for some $P' \subseteq P$, again since the templates do not change. We hence need to show:

If $D, \tau \vdash c \xrightarrow{\epsilon} c'$ and $\Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P\rangle$ then $\Delta, \emptyset, \emptyset \vdash c' : \mathrm{Clause}\langle P'\rangle$ for some $P' \subseteq P$.

The proof is by induction on the derivation of $D, \tau \vdash c \xrightarrow{\epsilon} c'$. We do a case split on the last derivation rule:

- The last rule is:

$$\frac{}{D, \tau \vdash \textbf{fulfilment} \xrightarrow{\epsilon} \textbf{fulfilment}}$$

  This case is trivial. (Note that $P = P' = \emptyset$.)
- The last rule is:

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau_1 \le \tau' \le \tau_2}{D, \tau \vdash \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1 \xrightarrow{(\tau', k(\vec{v}))} c_1[\vec{v}/\vec{x}, \tau_2 - \tau'/z]}$$

  The typing derivation for c has the form

$$\frac{\begin{matrix} & \emptyset \vdash p : \{p\} & \\ \Gamma' = [\vec{x} \mapsto \mathrm{ar}(k)] & \Gamma' \vdash e : \mathrm{Bool} & \overbrace{\hphantom{\Delta, \emptyset, \Gamma_2 \vdash c_1 : \mathrm{Clause}\langle P_2\rangle}}^{(a)} \\ \Gamma_2 = \Gamma'[z \mapsto \mathrm{Int}] & \emptyset \vdash d : \mathrm{Deadline} & \Delta, \emptyset, \Gamma_2 \vdash c_1 : \mathrm{Clause}\langle P_2\rangle \end{matrix}}{\Delta, \emptyset, \emptyset \vdash \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1 : \mathrm{Clause}\langle \{p\} \cup P_2\rangle}$$

It then follows from (a) and Lemma 9 that $\Delta, \emptyset, \emptyset \vdash c_1[\vec{v}/\vec{x}, \tau_2 - \tau'/z] : \mathrm{Clause}\langle P_2\rangle$, as required. (Note also that $P_2 \subseteq \{p\} \cup P_2$.)

- The last rule is:

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \quad \tau' \le \tau_2 \quad \tau' < \tau_1 \vee k' \neq k \vee e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \quad d' = \textbf{after}\ \tau_1 - \tau'\ \textbf{within}\ \tau_2 - \tau_1}{D, \tau \vdash \begin{matrix} \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1 \xrightarrow{(\tau', k'(\vec{v}))} \\ \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d'\ \textbf{remaining}\ z\ \textbf{then}\ c_1 \end{matrix}}$$

  We only need to show that $\emptyset \vdash d' : \mathrm{Deadline}$, which follows immediately.
- The last rule is

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau_1 \le \tau' \le \tau_2}{D, \tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau', k(\vec{v}))} c_1[\vec{v}/\vec{x}, \tau_2 - \tau'/z]}$$

  This case is similar to the second case.
- The last rule is:

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau' > \tau_2 \qquad \overbrace{D, \max(\tau, \tau_2) \vdash c_2 \xrightarrow{(\tau', k'(\vec{v}))} c'}^{(a)}}{D, \tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau', k'(\vec{v}))} c'}$$

  The typing derivation for c has the form

$$\frac{\begin{matrix} \Gamma' = [\vec{x} \mapsto \mathrm{ar}(k)] & \Gamma' \vdash e : \mathrm{Bool} & \overbrace{\Delta, \emptyset, \emptyset \vdash c_2 : \mathrm{Clause}\langle P_2\rangle}^{(b)} \\ \Gamma_1 = \Gamma'[z \mapsto \mathrm{Int}] & \emptyset \vdash d : \mathrm{Deadline} & \Delta, \emptyset, \Gamma_1 \vdash c_1 : \mathrm{Clause}\langle P_1\rangle \end{matrix}}{\Delta, \emptyset, \emptyset \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \mathrm{Clause}\langle P_1 \cup P_2\rangle}$$

  So the result follows from the induction hypothesis applied to (a) and (b), and from the fact that $P_2 \subseteq P_1 \cup P_2$.

- The last rule is:

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau' \leq \tau_2 \quad \tau' < \tau_1 \vee k' \neq k \vee e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \quad d' = \textbf{after}\ \tau_1 - \tau'\ \textbf{within}\ \tau_2 - \tau_1}{D, \tau \vdash \begin{array}{l} \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau', k'(\vec{v}))} \\ \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d'\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \end{array}}$$

  This case is similar to the third case.
- The last rule is:

$$\frac{\overbrace{D, \tau \vdash c_1 \xrightarrow{\epsilon} c_1'}^{(a)} \qquad \overbrace{D, \tau \vdash c_2 \xrightarrow{\epsilon} c_2'}^{(b)}}{D, \tau \vdash c_1\ \textbf{and}\ c_2 \xrightarrow{\epsilon} c_1'\ \textbf{and}\ c_2'}$$

  The typing derivation for c has the form

$$\frac{\overbrace{\Delta, \emptyset, \emptyset \vdash c_1 : \text{Clause}\langle P_1 \rangle}^{(c)} \qquad \overbrace{\Delta, \emptyset, \emptyset \vdash c_2 : \text{Clause}\langle P_2 \rangle}^{(d)}}{\Delta, \emptyset, \emptyset \vdash c_1\ \textbf{and}\ c_2 : \text{Clause}\langle P_1 \cup P_2 \rangle}$$

  So it follows from the induction hypothesis applied to (a) and (c) on one hand, and (b) and (d) on the other hand, that $\Delta, \emptyset, \emptyset \vdash c_1' : \text{Clause}\langle P_1' \rangle$ and $\Delta, \emptyset, \emptyset \vdash c_2' : \text{Clause}\langle P_2' \rangle$ with $P_1' \subseteq P_1$ and $P_2' \subseteq P_2$. Hence it follows that $\Delta, \emptyset, \emptyset \vdash c_1'\ \textbf{and}\ c_2' : \text{Clause}\langle P_1' \cup P_2' \rangle$ as required.
- The last rule is:

$$\frac{D, \tau \vdash c_1 \xrightarrow{\epsilon} c_1' \qquad D, \tau \vdash c_2 \xrightarrow{\epsilon} c_2'}{D, \tau \vdash c_1\ \textbf{or}\ c_2 \xrightarrow{\epsilon} c_1'\ \textbf{or}\ \mathbf{c}_2'}$$

  This case is similar to the previous case.
- The last rule is:

$$\frac{e \Downarrow \textbf{true} \qquad \overbrace{D, \tau \vdash c_1 \xrightarrow{\epsilon} c_1'}^{(a)}}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{\epsilon} c_1'}$$

  The typing derivation for c has the form

$$\frac{\emptyset \vdash e : \text{Bool} \qquad \overbrace{\Delta, \emptyset, \emptyset \vdash c_1 : \text{Clause}\langle P_1 \rangle}^{(b)} \qquad \Delta, \emptyset, \emptyset \vdash c_2 : \text{Clause}\langle P_2 \rangle}{\Delta, \emptyset, \emptyset \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \text{Clause}\langle P_1 \cup P_2 \rangle}$$

  So it follows from the induction hypothesis applied to (a) and (b) that $\Delta, \emptyset, \emptyset \vdash c_1' : \text{Clause}\langle P_1' \rangle$ with $P_1' \subseteq P_1 \subseteq P_1 \cup P_2$ as required.
- The last rule is:

$$\frac{e \Downarrow \textbf{false} \qquad D, \tau \vdash c_2 \xrightarrow{\epsilon} c_2'}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{\epsilon} c_2'}$$

  This case is similar to the previous case.
- The last rule is:

$$\frac{\vec{e} \Downarrow \vec{v} \qquad (f(\vec{x})\langle \vec{y} \rangle = c') \in D \qquad \overbrace{D, \tau \vdash c'[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle \vec{p}/\vec{y} \rangle \xrightarrow{\epsilon} c''}^{(a)}}{D, \tau \vdash f(\vec{e})\langle \vec{p} \rangle \xrightarrow{\epsilon} c''}$$

The typing derivation for c has the form

$$\frac{\Delta(f) = (\langle t_1, \ldots, t_m\rangle, n) \qquad \overbrace{\forall i \in \{1, \ldots, m\}. \emptyset \vdash e_i : t_i}^{\text{(b)}} \qquad \forall i \in \{1, \ldots, n\}. \emptyset \vdash p_i : \{p_i\}}{\Delta, \emptyset, \emptyset \vdash f(e_1, \ldots, e_m)\langle p_1, \ldots, p_n\rangle : \mathrm{Clause}\langle\{p_1, \ldots, p_n\}\rangle}$$

and it follows from $\Delta \vdash D$ that $\Delta, \vec{y}, [\vec{x} \mapsto \vec{t}, \vec{y} \mapsto \overrightarrow{\mathrm{Party}}] \vdash c' : \mathrm{Clause}\langle\emptyset\rangle$. It then follows from Lemma 8 and (b) that $v_i \in [\![t_i]\!]$ for $i = 1, \ldots, m$, and hence via Lemma 9 that $\Delta, \emptyset, \emptyset \vdash c'[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle\vec{p}/\vec{y}\rangle : \mathrm{Clause}\langle\{p_1, \ldots, p_n\}\rangle$. But then the result follows from the induction hypothesis applied to (a). □

We need the following two auxiliary lemmas in order to prove Theorem 11.

Lemma 15. *Assume that* $\mathrm{Sub}(c) = \{c_1, \ldots, c_n\}$*, for clauses* $c, c_1, \ldots, c_n$*. Then* $\mathrm{Sub}(c[\theta]) = \{c_1[\theta], \ldots, c_2[\theta]\}$ *for all substitutions* θ*.*

Proof. The proof follows by straightforward structural induction on c. □

Lemma 16. *Let c be a well-typed clause $\Delta, \emptyset, \emptyset \vdash c : Clause\langle P\rangle$. Then $\Delta, \emptyset, \emptyset \vdash c' : Clause\langle P'\rangle$ for all* $c' \in \mathrm{Sub}(c)$ *with* $P' \subseteq P$*.*

Proof. The proof follows by straightforward structural induction on c (or, equivalently by induction on the typing derivation of $\Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P\rangle$). □

Proof of Theorem 11. We start with a needed definition. We say that a substitution θ is *type-preserving* with regard to a variable environment Γ, if $\mathrm{dom}(\theta) = \mathrm{dom}(\Gamma)$ and $\theta(x) \in [\![\Gamma(x)]\!]$, for any $x \in \mathrm{dom}(\theta)$.

Let $s =$ **letrec** D **in** c_0 **starting** τ_0 and assume that s is well-formed with parties P. That is $\Rightarrow_D$ is an acyclic relation, $\Delta \vdash D$, and $\Delta, \emptyset, \emptyset \vdash c_0 : \mathrm{Clause}\langle P\rangle$ for some template environment Δ.

Assume $D = \{(f(\vec{x})\langle\vec{y}\rangle = c_f) \mid f \in \mathcal{F}_D\}$ and let $\mathcal{C}_D = \{c_f \mid f \in \mathcal{F}_D\}$. We associate with c_0 a new template name $f_0 \notin \mathcal{F}_D$, and let $\mathcal{F}'_D = \mathcal{F}_D \cup \{f_0\}$ and $c_{f_0} = c_0$. We extend the relation $\Rightarrow_D$ from $\mathcal{F}_D$ to $\mathcal{F}'_D$ as expected: $f_0 \Rightarrow_D g$ if and only if there is a subclause $g(\vec{e}_1)\langle\vec{e}_2\rangle \in \mathrm{Sub}(c_0)$. Note that by definition there is no $g \in \mathcal{F}'_D$ such that $g \Rightarrow_D f_0$. Hence the extended relation $\Rightarrow_D$ is still acyclic. And, as $\Rightarrow_D$ is finite, $\Rightarrow_D$ is well-founded.

We let $P_f = \emptyset$ for any $f \in \mathcal{F}_D$ and $P_{f_0} = P$. As $\Delta \vdash D$, there are environments Λ_f, Γ_f such that $\Delta, \Lambda_f, \Gamma_f \vdash c_f : \mathrm{Clause}\langle P_f\rangle$ for all $f \in \mathcal{F}'_D$, with $\Lambda_{f_0} = \emptyset$ and $\Gamma_{f_0} = \emptyset$. We will show the following claim:

Claim: For any $f \in \mathcal{F}'_D$, for any clause $c = c'[\theta]\langle\theta'\rangle$, where $c' \in \mathrm{Sub}(c_f)$, θ' is a party substitution with $\mathrm{dom}(\theta') = \Lambda_f$, and θ is a type-preserving substitution with regard to Γ_f, the following statement holds:

For any event ϵ with $\mathrm{ts}(\epsilon) \geq \tau_0$ there is a unique residue $\mathbf{c}$ such that $D, \tau_0 \vdash c \xrightarrow{\epsilon} \mathbf{c}$. Moreover, if $\mathbf{c} = (\tau, B)$, then $\tau_0 \leq \tau \leq \mathrm{ts}(\epsilon)$ and $B \subseteq P_f \cup \mathrm{rng}(\theta')$.

Note that the result of the theorem then follows from the claim applied to f_0, the clause c_0, and empty (party) substitutions θ and θ'.

We proceed by a nested inductive argument: an (outer) well-founded induction on f and an (inner) structural induction on the clause c.

The following observation will be used in the proof: since $\Delta, \Lambda_f, \Gamma_f \vdash c_f : \mathrm{Clause}\langle P_f\rangle$ it follows from Lemma 9 that $\Delta, \emptyset, \emptyset \vdash c_f[\theta]\langle\theta'\rangle : \mathrm{Clause}\langle P_f \cup \mathrm{rng}(\theta')\rangle$. Hence from Lemmas 15 and 16 it follows that $\Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P'\rangle$ with $P' \subseteq P_f \cup \mathrm{rng}(\theta')$, so we may assume in each case that c is well-typed and closed.

- $c =$ **fulfilment**. (This is a base case for the inner induction.) The claim clearly holds in this case.
- $c = \langle p\rangle\ k(\vec{x})$ **where** e **due** d **remaining** z **then** c_1. Suppose $\epsilon = (\tau', k'(\vec{v}))$ for some $\tau' \geq \tau_0$ and some action $k'(\vec{v})$. As c is well-typed, it follows from Lemma 8 that there is a unique Boolean value b and timestamps τ_1, τ_2 such that $e[\vec{v}/\vec{x}] \Downarrow b$ and $d \Downarrow^{\tau_0} (\tau_1, \tau_2)$. We distinguish three cases:
  - $k = k'$, $b =$ **true**, and $\tau_1 \leq \tau' \leq \tau_2$. Then take $\mathbf{c} = c[\vec{v}/\vec{x}, \tau_2 - \tau'/z]$.
  - $\tau' > \tau_2$. Take $\mathbf{c} = (\max(\tau_0, \tau_2), \{p\})$. Clearly, $\tau_0 \leq \max(\tau_0, \tau_2) \leq \tau'$. And, by the observation above, we know that $\Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P'\rangle$, where $P' \subseteq P_f \cup \mathrm{rng}(\theta')$, hence $p \in P_f \cup \mathrm{rng}(\theta')$.
  - $\tau' \leq \tau_2$ and also $k \neq k'$, $b =$ **false**, or $\tau' < \tau_1$. Then take $\mathbf{c} = \langle p\rangle\ k(\vec{x})$ **where** e **due** d' **remaining** z **then** c with $d' =$ **after** $\tau_1 - \tau'$ **within** $\tau_2 - \tau_1$.

  In all three cases the residue $\mathbf{c}$ satisfies the claim.
- $c =$ **if** $k(\vec{x})$ **where** e **due** d **remaining** z **then** c_1 **else** c_2. Suppose that $\epsilon = (\tau', k'(\vec{v}))$ for some $\tau' \geq \tau_0$ and some action $k'(\vec{v})$. As c is well-typed, it follows from Lemma 8 that there is a unique Boolean value b and timestamps τ_1, τ_2

such that $e[\vec{v}/\vec{x}] \Downarrow b$ and $d \Downarrow^{\tau_0} (\tau_1, \tau_2)$. As for obligations, we distinguish the same three cases, only the following one having a different treatment:

– $\tau' > \tau_2$. By the definition of immediate subclauses, we have that $c_2 \in \mathrm{Sub}(c_f)$, hence by the inner induction hypothesis on c_2 there is a unique residue $\mathbf{c}$ such that $D, \max(\tau_0, \tau_2) \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}$, and if $\mathbf{c} = (\tau, B)$ then $\max(\tau_0, \tau_2) \leq \tau \leq \mathrm{ts}(\epsilon)$ and $B \subseteq P$. Clearly, the residue $\mathbf{c}$ satisfies the claim.

- $c = c_1$ **and** c_2. By the definition of immediate subclauses, we have that $c_1, c_2 \in \mathrm{Sub}(c_f)$, hence by the inner induction hypothesis on c_1 and c_2 we obtain that there are unique residues $\mathbf{c}_1$ and $\mathbf{c}_2$ such that $D, \tau_0 \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1$ and $D, \tau_0 \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2$. Moreover, if $\mathbf{c}_1 = (\tau_1, B_1)$ then $\tau_0 \leq \tau_1 \leq \mathrm{ts}(\epsilon)$ and $B_1 \subseteq P$, and if $\mathbf{c}_2 = (\tau_2, B_2)$ then $\tau_0 \leq \tau_2 \leq \mathrm{ts}(\epsilon)$ and $B_2 \subseteq P$.

  Let $\mathbf{c} = \mathbf{c}_1 \owedge \mathbf{c}_2$. If $\mathbf{c}_1 = (\tau_1, B_1)$ and $\mathbf{c}_2 = (\tau_2, B_2)$, then it follows from the definition of verdict conjunction that $\tau_0 \leq \tau \leq \mathrm{ts}(\epsilon)$ and $B \subseteq P$, where $\mathbf{c} = (\tau, B) = (\tau_1, B_1) \wedge (\tau_2, B_2)$. In the other cases (that is $\mathbf{c}_1$ or $\mathbf{c}_2$ or both being clauses) the residue $\mathbf{c}$ clearly satisfies the claim.
- $c = c_1$ **or** c_2. This case is similar to the previous one, but in the case where $D, \tau_0 \vdash c_1 \xrightarrow{\epsilon} (\tau_1, B_1)$ and $D, \tau_0 \vdash c_2 \xrightarrow{\epsilon} (\tau_2, B_2)$, we utilise the fact that s is well-formed to conclude that $B_1 = B_2 = \{p\}$, for some p (due to the typing rule for clause disjunctions), which guarantees that the verdict disjunction $(\tau_1, B_1) \vee (\tau_2, B_2)$ is well-defined.
- $c =$ **if** e **then** c_1 **else** c_2. As c is well-typed, it follows from Lemma 8 that there is a unique Boolean value b such that $e \Downarrow b$. By the definition of immediate subclauses, we have that $c_1, c_2 \in \mathrm{Sub}(c_f)$, hence by the inner induction hypothesis on c_1 if $b =$ **true** and on c_2 otherwise, the claim follows directly.
- $c = g(\vec{e})\langle\vec{p}\rangle$. As c is well-typed, it follows from Lemma 8 that there are unique values $\vec{v}$ such that $\vec{e} \Downarrow \vec{v}$. Moreover, by hypothesis the clause c is the instantiation of an immediate subclause $g(\vec{e}_1)\langle\vec{e}_2\rangle$ of c_f. By the definition of $\Rightarrow_D$, we have that $f \Rightarrow_D g$. This, together with $[\vec{v}/\vec{x}, \vec{p}/\vec{y}]$ being a type-preserving substitution with regard to Γ_g (Lemma 8) and $\langle\vec{p}/\vec{y}\rangle$ being a party substitution, allows us to apply the outer induction hypothesis on $c_g[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle\vec{p}/\vec{y}\rangle$. The claim then follows directly. □

We need the following auxiliary lemma in order to prove Theorem 12.

Lemma 17. *Let s be a well-formed specification. Then there exists a unique verdict ν such that $\vdash s \downarrow \nu$. Moreover, for a breach (τ, B), we have $\vdash s \downarrow (\tau, B)$ if and only if $s \xrightarrow{\epsilon} (\tau, B)$, for all events ϵ with $\mathrm{ts}(\epsilon) > \tau$.*

Proof. Existence follows by a nested inductive argument similar to, but much simpler than the proof of Theorem 11. Uniqueness follows by straightforward structural induction on c, where $s =$ **letrec** D **in** c **starting** τ. The left to right implication of the second part of the lemma follows by induction on the derivation of $\vdash s \downarrow (\tau, B)$, while the other implication follows by induction on the derivation of $s \xrightarrow{\epsilon} (\tau, B)$. □

Proof of Theorem 12. Let $s =$ **letrec** D **in** c **starting** τ_0 be a well-formed specification with parties P. We then need to show that $[\![s]\!]$ is a contract between P starting at time τ_0. That is we need to show that $[\![s]\!]$ is a function from Tr^{τ_0} to V, and that it satisfies conditions (1) and (2) of Definition 1.

We first prove by induction on the length of the *finite* trace σ that: $[\![s]\!](\sigma)$ is well-defined, that is it exists and it is unique, and if $[\![s]\!](\sigma) = (\tau, B)$ then $B \subseteq P$, $[\![s]\!](\sigma_\tau) = (\tau, B)$, and $\tau \geq \tau_0$.

Base case: $\sigma = \langle\rangle$. In this case it follows from Lemma 17 that there is a unique verdict ν such that $\vdash s \downarrow \nu$, and hence $[\![s]\!](\sigma) = \nu$. So assume now that $[\![s]\!](\sigma) = (\tau, B)$. Then since $\sigma_\tau = \sigma$ we also have that $[\![s]\!](\sigma_\tau) = (\tau, B)$. Lastly, it follows from Lemma 17 that $s \xrightarrow{\epsilon} (\tau, B)$, for any event ϵ with $\mathrm{ts}(\epsilon) > \max(\tau, \tau_0)$, and hence from Theorem 11 we have that $B \subseteq P$ and $\tau \geq \tau_0$ as required.

Inductive case: $\sigma = \epsilon\sigma'$. As s is well-formed and $\mathrm{ts}(\epsilon) \geq \tau_0$, it follows from the progress property (Theorem 11) that there is a unique residue $\mathbf{s}$ such that $s \xrightarrow{\epsilon} \mathbf{s}$.

- If $\mathbf{s} = (\tau, B)$ then, also from Theorem 11, we have that $B \subseteq P$ and $\tau_0 \leq \tau \leq \mathrm{ts}(\epsilon)$. Now, if $\mathrm{ts}(\epsilon) = \tau$ then $\sigma_\tau = \epsilon\sigma'_\tau$ so it follows immediately that $[\![s]\!](\sigma_\tau) = (\tau, B)$. So assume that $\mathrm{ts}(\epsilon) > \tau$. It then follows from Lemma 17 that $\vdash s \downarrow (\tau, B)$ and hence $[\![s]\!](\sigma_\tau) = (\tau, B)$ as required.
- If $\mathbf{s} = s'$ then, by the type-preservation property (Theorem 10), s' is also well-formed with parties $P' \subseteq P$ and s' has starting time $\mathrm{ts}(\epsilon)$. We have that $[\![s]\!](\sigma) = [\![s']\!](\sigma')$, so it then follows from the induction hypothesis that $[\![s']\!](\sigma')$ is well-defined and if $[\![s']\!](\sigma') = (\tau, B)$ then $B \subseteq P' \subseteq P$, $[\![s']\!](\sigma'_\tau) = (\tau, B)$, and $\tau_0 \leq \mathrm{ts}(\epsilon) \leq \tau$.

  Now if $[\![s]\!](\sigma) = (\tau, B)$ then $[\![s]\!](\epsilon\sigma') = [\![s']\!](\sigma') = (\tau, B)$ and hence by the above $[\![s']\!](\sigma'_\tau) = (\tau, B)$ with $\tau \geq \mathrm{ts}(\epsilon)$. But then $\sigma_\tau = \epsilon\sigma'_\tau$, and hence by definition $[\![s]\!](\sigma_\tau) = [\![s']\!](\sigma'_\tau) = (\tau, B)$ as required.

We now show that if $[\![s]\!](\sigma) = (\tau, B)$ for some finite trace σ and breach (τ, B), then $[\![s]\!](\sigma') = (\tau, B)$, for any finite trace σ' with $\sigma'_\tau = \sigma_\tau$. Let σ' be a trace with $\sigma'_\tau = \sigma_\tau$. As shown above, we have $[\![s]\!](\sigma_\tau) = (\tau, B)$. The proof is by induction on the length of σ_τ:

Base case: $\sigma_\tau = \langle\rangle$. Now $\sigma'_\tau = \langle\rangle$, so either $\sigma' = \langle\rangle$ or $\sigma' = \epsilon\sigma''$, for some ϵ and σ'' with $\mathrm{ts}(\epsilon) > \tau$. In the first case the result follows immediately, and in the second case we have that $\vdash s \downarrow (\tau, B)$, hence by Lemma 17 we have that $s \xrightarrow{\epsilon} (\tau, B)$ from which the result follows.

Inductive case: $\sigma_\tau = \epsilon\sigma''$. Now $\sigma' = \epsilon\sigma'''$ with $\sigma'' = \sigma'''_\tau$, and $[\![s]\!](\sigma_\tau) = (\tau, B)$ can happen in two ways:

- $s \xrightarrow{\epsilon} (\tau, B)$: In this case we have by definition that $[\![s]\!](\sigma') = [\![s]\!](\epsilon\sigma''') = (\tau, B)$.
- $s \xrightarrow{\epsilon} s'$ and $[\![s']\!](\sigma'') = (\tau, B)$: In this case we have by definition that $[\![s]\!](\sigma') = [\![s']\!](\sigma''')$, and hence the result follows from the induction hypothesis as $\sigma'' = \sigma'''_\tau$.

We have now proved that the restriction of $[\![s]\!]$ on finite traces satisfies the hypotheses of Lemma 3. We can thus apply the lemma and obtain that $[\![s]\!]$ is a contract as per Definition 1. $\square$

References

[1] B. Alpern, F.B. Schneider, Defining liveness, Inform. Process. Lett. 21 (4) (1985) 181–185.
[2] J. Andersen, E. Elsborg, F. Henglein, J.G. Simonsen, C. Stefansen, Compositional specification of commercial contracts, Int. J. Softw. Tools Technol. Transfer 8 (6) (2006) 485–516.
[3] A. Boulmakoul, M. Sallé, Integrated contract management, Tech. Rep. HPL-2002-183, HP Laboratories Bristol, Bristol, United Kingdom, 2002.
[4] Content Reference Forum, Contract Expression Language (CEL) – an UN/CEFACT BCF Compliant Technology, January 2004.
[5] J.W. Forrester, Gentle murder, or the adverbial samaritan, J. Philos. 81 (4) (1984) 193–197.
[6] J.-Y. Girard, Linear logic, Theoret. Comput. Sci. 50 (1987) 1–102.
[7] A. Goodchild, C. Herring, Z. Milosevic, Business contracts for B2B, in: Proceedings of the CAiSE 2000 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing (ISDO), CEUR Workshop Proceedings, vol. 30, 2000, pp. 63–74.
[8] G. Governatori, Representing business contracts in RuleML, Int. J. Cooperative Inform. Syst. 14 (2005) 181–216.
[9] G. Governatori, Z. Milosevic, A formal analysis of a business contract language, Int. J. Cooperative Inform. Syst. 15 (4) (2006) 659–685.
[10] G. Governatori, D.H. Pham, DR-CONTRACT: an architecture for e-contracts in defeasible logic, Int. J. Bus. Process Integration Management 4 (3) (2009) 187–199.
[11] M. Kyas, C. Prisacariu, G. Schneider, Run-time monitoring of electronic contracts, in: Proceedings of the Sixth International Symposium on Automated Technology for Verification and Analysis (ATVA), Lecture Notes Computer Science, vol. 5311, Springer, Heidelberg, Germany, 2008, pp. 397–407.
[12] R.M. Lee, A logic model for electronic contracting, Decision Support Syst. 4 (1) (1988) 27–44.
[13] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Logic Algebr. Programming 78 (5) (2009) 293–303.
[14] P.F. Linington, Z. Milosevic, J.B. Cole, S. Gibson, S. Kulkarni, S.W. Neal, A unified behavioural model and a contract language for extended enterprise, Data Knowledge Eng. 51 (1) (2004) 5–29.
[15] Microsoft Dynamics AX, 2011. Available from: <http://www.microsoft.com/en-us/dynamics/products/ax-overview.aspx>.
[16] Microsoft Dynamics NAV, 2011. Available from: <http://www.microsoft.com/en-us/dynamics/products/nav-overview.aspx>.
[17] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, J. Warne, Run-time monitoring and enforcement of electronic contracts, Electron. Commerce Res. Appl. 3 (2) (2004) 108–125.
[18] N. Oren, S. Panagiotidi, J. Vázquez-Salceda, S. Modgil, M. Luck, S. Miles, Towards a formalisation of electronic contracting environments, Revised Selected Papers of the 2008 International Workshops on Coordination, Organizations, Institutions and Norms in Agent Systems (COIN), Lecture Notes Computer Science, vol. 5428, Springer, Heidelberg, Germany, 2009, pp. 156–171.
[19] G.J. Pace, G. Schneider, Challenges in the specification of full contracts, in: Proceedings of the Seventh International Conference on Integrated Formal Methods (IFM), Lecture Notes Computer Science, vol. 5423, Springer, Heidelberg, Germany, 2009, pp. 292–306.
[20] V. Patel, The contract management benchmark report: procurement contracts, Tech. Rep., Aberdeen Group, Boston, MA, USA, 2006.
[21] V. Patel, C.J. Dwyer, Contract lifecycle management and the cfo: optimizing revenues and capturing savings, Tech. Rep., Springer, Boston, MA, USA, 2007.
[22] S. Peyton-Jones, J.-M. Eber, How to write a financial contract, in: J. Gibbons, O. de Moor (Eds.), The Fun of Programming, Palgrave Macmillan Ltd., London, United Kingdom, 2003, pp. 105–130 (Chapter 6).
[23] B.C. Pierce, Advanced Topics in Types and Programming Languages, The MIT Press, Cambridge, MA, USA, 2005.
[24] B.C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, USA, 2002.
[25] H. Prakken, M. Sergot, Contrary-to-duty obligations, Studia Logica 57 (1996) 91–115.
[26] C. Prisacariu, G. Schneider, A formal language for electronic contracts, in: Proceedings of the Nineth IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Lecture Notes Computer Science, vol. 4468, Springer, Heidelberg, Germany, 2007, pp. 174–189.
[27] Y.-H. Tan, W. Thoen, INCAS: a legal expert system for contract terms in electronic commerce, Decision Support Syst. 29 (4) (2000) 389–411.
[28] G.H. von Wright, Deontic logic, Mind 60 (237) (1951) 1–15.
[29] H. Weigand, L. Xu, Contracts in e-commerce, in: Proceedings of the IFIP TC2/WG2.6 Nineth Working Conference on Database Semantics, IFIP Conference Proceedings, vol. 239, Kluwer, Deventer, The Netherlands, 2003, pp. 3–17.
[30] J. Woleński, Deontic logic and possible worlds semantics: a historical sketch, Studia Logica 49 (2) (1990) 273–282.
[31] L. Xu, A multi-party contract model, SIGecom Exchange 5 (1) (2004) 13–23.
[32] L. Xu, M.A. Jeusfeld, Pro-active monitoring of electronic contracts, in: Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE), Lecture Notes Computer Science, vol. 2681, Springer, Heidelberg, Germany, 2003, pp. 584–600.