Run-time Monitoring of Electronic Contracts

Marcel Kyas*, Cristian Prisacariu**, and Gerardo Schneider**

Department of Informatics, University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
{kyas,cristi,gerardo}@ifi.uio.no

Abstract. Electronic inter-organizational relationships are governed by contracts regulating their interaction. It is necessary to run-time monitor the contracts, as to guarantee their fulfillment. The present work shows how to obtain a run-time monitor for contracts written in $\mathcal{CL}$, a formal specification language which allows to write conditional obligations, permissions and prohibitions over actions. The trace semantics of $\mathcal{CL}$ formalizes the notion of *a trace fulfills a contract*. We show how to obtain, for a given contract, an alternating Büchi automaton which accepts exactly the traces that fulfill the contract. This automaton is the basis for obtaining a finite state machine which acts as a run-time monitor for $\mathcal{CL}$ contracts.

1 Introduction

Internet inter-business collaborations, virtual organizations, and web services, usually communicate through service exchanges which respect an implicit or explicit *contract*. Such a contract must unambiguously determine correct interactions, and what are the exceptions allowed, or penalties imposed in case of incorrect behavior.

Legal contracts, as found in the usual judicial or commercial arena, may serve as basis for defining such machine-oriented *electronic contracts* (or e-contracts for short). Ideally, e-contracts should be shown to be contradiction-free both internally, and with respect to the governing policies under which the contract is enacted. Moreover, there must be a run-time system ensuring that the contract is respected. In other words, contracts should be amenable to formal analysis allowing both static and dynamic verification, and therefore written in a formal language. In this paper we are interested in the run-time monitoring of electronic contracts, and not in the static verification of their consistency or conformance with policies.

$\mathcal{CL}$, introduced in [12], is an action-based formal language tailored for writing e-contracts, with the following properties: (1) Avoids most of the common

* Author supported by the EU-project IST-33826 *"CREDO: Modelling and analysis of evolutionary structures for distributed services"* (http://credo.cwi.nl).

** Authors supported by the Nordunet3 project *"COSoDIS – Contract-Oriented Software Development for Internet Services"* (http://www.ifi.uio.no/cosodis/).

$$
\begin{array}{rcl}
\mathcal{C} & := & \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\beta]\mathcal{C} \mid \top \mid \ \bot \\
\mathcal{C}_O & := & O_{\mathcal{C}}(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P & := & P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F & := & F_{\mathcal{C}}(\alpha) \mid \mathcal{C}_F \vee [\alpha]\mathcal{C}_F \\
\alpha & := & \mathbf{0} \mid \mathbf{1} \mid a \mid \alpha \& \alpha \mid \alpha \cdot \alpha \mid \alpha + \alpha \\
\beta & := & \mathbf{0} \mid \mathbf{1} \mid a \mid \beta \& \beta \mid \beta \cdot \beta \mid \beta + \beta \mid \beta^* \mid \mathcal{C}?
\end{array}
$$

Table 1. Syntax of the $\mathcal{CL}$ language to use for specifying contracts.

philosophical paradoxes of deontic logic [18]; (2) Has a formal semantics given in terms of Kripke structures [13]; (3) It can express (conditional) obligations, permission and prohibition over concurrent actions; as well as (4) *contrary-to-duty obligations* (CTD) and *contrary-to-prohibitions* (CTP). CTDs/CTPs specify the obligation/prohibition to be fulfilled and which is the *reparation/penalty* to be applied in case of violation. The use of e-contracts, and in particular of $\mathcal{CL}$, goes beyond the application domain of service-exchanges, comprising component-based development systems, fault-tolerant and embedded systems.

The main contribution of this paper is an automatic procedure for obtaining a run-time monitor for contracts, directly extracted from the $\mathcal{CL}$ specification. The road-map of the paper starts by recalling main results on $\mathcal{CL}$ in first part of Section 2 and we give a trace semantics for the expressions of $\mathcal{CL}$ in the second part. This expresses the fact that a trace respects (does not violate) a contract clause (expression of $\mathcal{CL}$). In Section 3 we show how to construct for a contract an alternating Büchi automaton which recognizes exactly all the traces respecting the contract. The automaton is used in Section 4 for constructing the monitor as a Moore machine (for monitoring the contract). Though we concentrate on theoretical aspects, we use throughout the paper the following small didactic example to exemplify some of the main concepts we define.

Example 1. "If the *Client* exceeds the bandwidth limit then (s)he must pay [*price*] immediately, or (s)he must delay the payment and notify the *Provider* by sending an e-mail. If in breach of the above (s)he must pay double."

2 $\mathcal{CL}$ – A Formal Language for Contracts

$\mathcal{CL}$ is an *action-based* language for writing contracts [12]. In this paper we are interested in monitoring the actions of a contract. Therefore, we give here a slightly different version of $\mathcal{CL}$ where we have dropped the assertions from the old $\mathcal{CL}$, keeping only the modalities over actions. Other differences are in the expressivity: we have incorporated the Kleene star operator over the actions in the dynamic box modality, and we have attached to the obligations the corresponding reparations (modelling the CTDs directly).

Syntax: $\mathcal{CL}$ formulas are defined by the grammar in Table 1. In what follows we provide intuitions for the $\mathcal{CL}$ syntax and define our notation and terminology.

We call an expression $\mathcal{C}$ a (general) *contract clause*. $\mathcal{C}_O$, $\mathcal{C}_P$, and $\mathcal{C}_F$ are called respectively *obligation*, *permission*, and *prohibition* clauses. We call $O_{\mathcal{C}}(\alpha)$, $P(\alpha)$, and $F_{\mathcal{C}}(\alpha)$ the *deontic modalities*, and they represent the obligation, permission,

or prohibition of performing a given *action* α. Intuitively $O_{\mathcal{C}}(\alpha)$ states the obligation to execute α, and the *reparation* $\mathcal{C}$ in case the obligation is *violated*, i.e. whenever α is not performed.[1] The reparation may be any contract clause. Obligations without reparations are written as $O_{\bot}(\alpha)$ where $\bot$ (and conversely $\top$) is the Boolean `false` (respectively `true`). We usually write $O(\alpha)$ instead of $O_{\bot}(\alpha)$. The prohibition modality $F_{\mathcal{C}}(\alpha)$ states the actual forbearing of the action $F(\alpha)$ together with the reparation $\mathcal{C}$ in case the prohibition is violated. Note that it is possible to express nested CTDs and CTPs.

Throughout the paper we denote by $a, b, c \in \mathcal{A}_B$ the *basic actions*, by indexed $\alpha \in \mathcal{A}$ *compound actions*, and by indexed β the actions found in propositional dynamic logic [4] with intersection [5]. Actions α are used inside the deontic modalities, whereas the (more general) actions β are used inside the dynamic modality. An *action* is an expression containing one or more of the following binary constructors: *choice* "+", *sequence* "$\cdot$", *concurrency* "&" and are constructed from the basic actions $a \in \mathcal{A}_B$ and $\mathbf{0}$ and $\mathbf{1}$ (called the *violating* action and respectively *skip* action). Indepth reading and results related to the α actions can be found in [13]. Actions β have the extra operators Kleene star * and *test* $?$.[2] To avoid using parentheses we give a precedence over the constructors: $\& > \cdot > +$. *Concurrent actions*, denoted by $\alpha_{\&}$, are actions of $\mathcal{A}_B^{\&} \subset \mathcal{A}$ generated from basic actions using only the & constructor (e.g. a, $a\&a, a\&b \in \mathcal{A}_B^{\&}$ and $a + b, a\&b + c, a \cdot b \notin \mathcal{A}_B^{\&}$). Note that $\mathcal{A}_B^{\&}$ is finite because $\mathcal{A}_B$ is defined as finite and & is defined idempotent over basic actions. Therefore, we consider concurrent actions of $\mathcal{A}_B^{\&}$ as sets over basic actions of $\mathcal{A}_B$. We have now a natural way to compare concurrent actions using $\subseteq$ set inclusion. We say that an action, e.g. $a \,\&\, b \,\&\, c$ is *more demanding* than another action, e.g. $a \,\&\, b$ iff $\{a, b\} \subseteq \{a, b, c\}$. The *negation* $\overline{\alpha}$ of action α is a function $\overline{\ } : \mathcal{A} \to \mathcal{A}$.

We use the propositional operators $\wedge$, $\vee$, and $\oplus$ (exclusive or). The dynamic logic modality $[\cdot]\mathcal{C}$ is parameterized by *actions* β. The expression $[\beta]\mathcal{C}$ states that after the action β is performed $\mathcal{C}$ must hold. The $[\cdot]$ modality allows having a *test* inside, and $[\mathcal{C}_1?]\mathcal{C}_2$ must be understood as $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$. In $\mathcal{CL}$ we can write *conditional* obligations (permissions and prohibitions) of two kinds: $[\beta]O(\alpha)$ read as "after performing β, one is obliged to do α", and using the test operator $[\mathcal{C}?]O(\alpha)$ to simulate implication. Similarly for permission and prohibition.

Example 1 in $\mathcal{CL}$ syntax: The transition from the conventional contract of introduction to the $\mathcal{CL}$ expression below is manual.

$$[e]O_{O_{\bot}(p \cdot p)}(p + d\&n)$$

where the basic actions are $\mathcal{A}_B = \{e, p, n, d\}$ (standing for "extend bandwidth limit", "pay", "notify by email", and "delay"). In short the expression is read as: After executing the action e there is the obligation of choosing between either p or at the same time d and n. The $\mathcal{CL}$ expression also states the reparation $O_{\bot}(p \cdot p)$

[1] The modality $O_{\mathcal{C}}(\alpha)$ (resp. $F_{\mathcal{C}}(\alpha)$) represents what is called CTD (resp. CTP) in the deontic logic community.

[2] The investigation of the PDL actions β can be found in the literature related to dynamic and Kleene algebras [7].

$\sigma \models O_{\mathcal{C}}(\alpha_{\&})$ if $\alpha_{\&} \subseteq \sigma(0)$, or if $\sigma(1..) \models \mathcal{C}$.
$\sigma \models O_{\mathcal{C}}(\alpha \cdot \alpha')$ if $\sigma \models O_{\mathcal{C}}(\alpha)$ and $\sigma \models [\alpha] O_{\mathcal{C}}(\alpha')$.
$\sigma \models O_{\mathcal{C}}(\alpha + \alpha')$ if $\sigma \models O_{\perp}(\alpha)$ or $\sigma \models O_{\perp}(\alpha')$ or $\sigma \models [\overline{\alpha + \alpha'}]\mathcal{C}$.
$\sigma \models F_{\mathcal{C}}(\alpha_{\&})$ if $\alpha_{\&} \not\subseteq \sigma(0)$, or if $\alpha_{\&} \subseteq \sigma(0)$ and $\sigma(1..) \models \mathcal{C}$.
$\sigma \models F_{\mathcal{C}}(\alpha \cdot \alpha')$ if $\sigma \models F_{\perp}(\alpha)$ or $\sigma \models [\alpha] F_{\mathcal{C}}(\alpha')$.
$\sigma \models F_{\mathcal{C}}(\alpha + \alpha')$ if $\sigma \models F_{\mathcal{C}}(\alpha)$ and $\sigma \models F_{\mathcal{C}}(\alpha')$.
$\sigma \models [\overline{\alpha_{\&}}]\mathcal{C}$ if $\alpha_{\&} \not\subseteq \sigma(0)$ and $\sigma(1..) \models \mathcal{C}$, or if $\alpha_{\&} \subseteq \sigma(0)$.
$\sigma \models [\overline{\alpha \cdot \alpha'}]\mathcal{C}$ if $\sigma \models [\overline{\alpha}]\mathcal{C}$ and $\sigma \models [\alpha][\overline{\alpha'}]\mathcal{C}$.
$\sigma \models [\overline{\alpha + \alpha'}]\mathcal{C}$ if $\sigma \models [\overline{\alpha}]\mathcal{C}$ or $\sigma \models [\overline{\alpha'}]\mathcal{C}$.

Table 2. Trace semantics of $\mathcal{CL}$. Dynamic and propositional operators are omitted [6].

in case the obligation above is violated which is an obligation of doing twice in a row the action of paying. Note that this second obligation has no reparation attached, therefore if it is violated then the whole contract is violated. Note also that we translate "pay double" into the $\mathcal{CL}$ sequential composition of the same action p of paying.

Semantics on Respecting Traces: The rest of this section is devoted to presenting a semantics for $\mathcal{CL}$ with the goal of monitoring electronic contracts. For this we are interested in identifying the traces of actions which are *respecting* or *violating* a contract clause. We follow the many works in the literature which have a presentation based on traces e.g. [11].

Definition 1 (traces). *Consider a* trace *denoted $\sigma = a_0, a_1, \ldots$ as an ordered sequence of concurrent actions. Formally a trace is a map $\sigma : \mathbb{N} \to \mathcal{A}_B^{\&}$ from natural numbers (denoting positions) to concurrent actions from $\mathcal{A}_B^{\&}$. Take $m_\sigma \in \mathbb{N} \cup \infty$ to be the length of a trace. A (infinite) trace which from some position m_σ onwards has only action* $\mathbf{1}$ *is considered* finite. *We use ε to denote the* empty trace. *We denote by $\sigma(i)$ the* element *of a trace at position i, by $\sigma(i..j)$ a finite* subtrace, *and by $\sigma(i..)$ the infinite subtrace starting at position i in σ. The* concatenation *of two traces σ' and σ'' is denoted $\sigma'\sigma''$ and is defined iff the trace σ' is finite; $\sigma'\sigma''(i) = \sigma'(i)$ if $i < m_{\sigma'}$ and $\sigma'\sigma''(i) = \sigma''(i - m_{\sigma'})$ for $i \geq m_{\sigma'}$ (e.g. $\sigma(0)$ is the first action of a trace, $\sigma = \sigma(0..i)\sigma'$ where $\sigma' = \sigma(i{+}1..)$).*

Definition 2 (Semantics of $\mathcal{CL}$). *We give in Table 2 a recursive definition of the* satisfaction relation $\models$ *over pairs $(\sigma, \mathcal{C})$ of traces and contracts; it is usually written $\sigma \models \mathcal{C}$ and we read it as "trace σ* respects *the contract (clause) $\mathcal{C}$". We write $\sigma \not\models \mathcal{C}$ instead of $(\sigma, \mathcal{C}) \notin \models$ and read it as "σ* violates *$\mathcal{C}$."*

A trace σ respects an obligation $O_{\mathcal{C}}(\alpha_{\&})$ if either of the two complementary conditions is satisfied. The first condition deals with the obligation itself: the trace σ respects the obligation $O(\alpha_{\&})$ if the first action of the trace includes $\alpha_{\&}$. Otherwise, in case the obligation is violated,[3] the only way to fulfill the contract

[3] Violation of an obligatory action is encoded by the action negation.

is by respecting the reparation $\mathcal{C}$; i.e. $\sigma(1..) \models \mathcal{C}$. Respecting an obligation of a choice action $O_{\mathcal{C}}(\alpha_1 + \alpha_2)$ means that it must be executed one of the actions α_1 or α_2 completely; i.e. obligation needs to consider only one of the choices. If none of these is entirely executed then a violation occurs (thus the negation of the action is needed) so the reparation $\mathcal{C}$ must be respected. An important requirement when modelling electronic contracts is that the obligation of a sequence of actions $O_{\mathcal{C}}(\alpha \cdot \alpha')$ must be equal to the obligation of the first action $O_{\mathcal{C}}(\alpha)$ and after the first obligation is respected the second obligation must hold $[\alpha]O_{\mathcal{C}}(\alpha')$. Note that if $O_{\mathcal{C}}(\alpha)$ is violated then it is required that the second obligation is discarded, and the reparation $\mathcal{C}$ must hold. Violating $O_{\mathcal{C}}(\alpha)$ means that α is not executed and thus, by the semantic definition, $[\alpha]O_{\mathcal{C}}(\alpha')$ holds regardless of $O_{\mathcal{C}}(\alpha')$.

From [6] we know how to encode LTL only with the dynamic $[\cdot]$ modality and the Kleene *; e.g. "*always* obliged to do α" is encoded as $[\mathbf{any}^*]O(\alpha)$ where $\mathbf{any} \triangleq +_{\gamma \in \mathcal{A}^{\&}_{B}} \gamma$ is the choice between *any* concurrent action.

3 Satisfiability checking using alternating automata

Automata theoretic approach to satisfiability of temporal logics was introduced in [17] and has been extensively used and developed since. We recall first basic theory of automata on infinite objects. We follow the presentation and use the notation of Vardi [16]. Given an alphabet Σ, a *word over* Σ is a sequence $a_0, a_1 \ldots$ of symbols from Σ. The set of *infinite words* is denoted by Σ^ω.

We denote by $\mathcal{B}^+(X)$ the set of positive Boolean formulas θ (i.e. containing only $\wedge$ and $\vee$, and not the $\neg$) over the set X together with the formulas **true** and **false**. For example $\theta = (s_1 \vee s_2) \wedge (s_3 \vee s_4)$ where $s_i \in X$. A subset $Y \subseteq X$ is said to *satisfy* a formula θ iff the truth assignment which assigns *true* only to the elements of Y assigns *true* also to θ. In the example, the set $\{s_1, s_3\}$ satisfies θ; but this set is not unique.

An *alternating Büchi automaton* [2,10] is a tuple $A = (S, \Sigma, s_0, \rho, F)$, where S is a finite nonempty set of *states*, Σ is a finite nonempty *alphabet*, $s_0 \in S$ is the *initial* state, and $F \subseteq S$ is the set of *accepting* states. The automaton can move from one state when it reads a symbol from Σ according to the transition function $\rho : S \times \Sigma \to \mathcal{B}^+(S)$. For example $\rho(s_0, a) = (s_1 \vee s_2) \wedge (s_3 \vee s_4)$ means that the automaton moves from s_0 when reading a to state s_1 or s_2 and at the same time to state s_3 or s_4. Intuitively the automaton chooses for each transition $\rho(s, a) = \theta$ one set $S' \in S$ which satisfies θ and spawns a copy of itself for each state $s_i \in S'$ which should test the acceptance of the remaining word from that state s_i. Alternating automata combine existential choice of nondeterministic finite automata (i.e. disjunction) with the universal choice (i.e. conjunction) of $\forall$-automata [9] (where from a state the automaton must move to *all* the next states given by the transition function).

Because the alternating automaton moves to all the states of a (nondeterministically chosen) satisfying set of θ, a run of the automaton is a *tree* of states. Formally, a run of the alternating automaton on an input word $\alpha = a_0, a_1, \ldots$ is an S-labeled tree $(T, \mathcal{V})$ (i.e. the nodes of the tree are labeled by $\mathcal{V}$ with state names of the automaton) such that $\mathcal{V}(\varepsilon) = s_0$ and the following hold:

$$
\begin{aligned}
FL(\top) &\triangleq \{\top\} \qquad FL(\bot) \triangleq \{\bot\} \qquad FL(P(\alpha)) \triangleq \{P(\alpha)\} \\
FL(O_{\mathcal{C}}(\alpha_{\&})) &\triangleq \{O_{\mathcal{C}}(\alpha_{\&})\} \cup FL(\mathcal{C}) \\
FL(O_{\mathcal{C}}(\alpha \cdot \alpha')) &\triangleq \{O_{\mathcal{C}}(\alpha \cdot \alpha')\} \cup FL(O_{\mathcal{C}}(\alpha)) \cup FL([\alpha]O_{\mathcal{C}}(\alpha')) \\
FL(O_{\mathcal{C}}(\alpha + \alpha')) &\triangleq \{O_{\mathcal{C}}(\alpha + \alpha')\} \cup FL(O_{\bot}(\alpha)) \cup FL(O_{\bot}(\alpha')) \cup FL(\mathcal{C}) \\
FL(F_{\mathcal{C}}(\alpha_{\&}) &\triangleq \{F_{\mathcal{C}}(\alpha_{\&}\} \cup FL(\mathcal{C}) \\
FL(F_{\mathcal{C}}(\alpha \cdot \alpha')) &\triangleq \{F_{\mathcal{C}}(\alpha \cdot \alpha')\} \cup FL(F_{\bot}(\alpha)) \cup FL(F_{\mathcal{C}}(\alpha')) \\
FL(F_{\mathcal{C}}(\alpha + \alpha')) &\triangleq \{F_{\mathcal{C}}(\alpha + \alpha')\} \cup FL(F_{\mathcal{C}}(\alpha)) \cup FL(F_{\mathcal{C}}(\alpha'))
\end{aligned}
$$

Table 3. Computing the Fisher-Ladner Closure (see [6] for the standard operators).

for a node x with $|x| = i$ s.t. $\mathcal{V}(x) = s$ and $\rho(s, a_i) = \theta$ then x has k children $\{x_1, \ldots, x_k\}$ which is the number of states in the chosen satisfying set of states of θ, say $\{s_1, \ldots, s_k\}$, and the children are labeled by the states in the satisfying set; i.e. $\{\mathcal{V}(x_1) = s_1, \ldots, \mathcal{V}(x_k) = s_k\}$.

For example, if $\rho(s_0, a) = (s_1 \vee s_2) \wedge (s_3 \vee s_4)$ then the nodes of the run tree at the first level have one label among s_1 or s_2 and one label among s_3 or s_4. When $\rho(\mathcal{V}(x), a) = \mathbf{true}$, then x need not have any children; i.e. the branch reaching x is finite and ends in x. A run tree of an alternating Büchi automaton is *accepting* if every infinite branch of the tree includes infinitely many nodes labeled by accepting states of F. Note that the run tree may also have finite branches in the cases when the transition function returns **true**.

Fischer-Ladner closure for $\mathcal{CL}$: For constructing the alternating automaton for a $\mathcal{CL}$ expression we need the Fischer-Ladner closure [4] for our $\mathcal{CL}$ logic. We follow the presentation in [6] and use similar terminology. We define a function $FL : \mathcal{CL} \to 2^{\mathcal{CL}}$ which for each expression $\mathcal{C}$ of the logic $\mathcal{CL}$ returns the set of its subexpressions. The function FL is defined inductively in Table 3 (see also [14]).

Theorem 1 (automaton construction). *Given a $\mathcal{CL}$ expression $\mathcal{C}$, one can build an alternating Büchi automaton $A^{\mathcal{N}}(\mathcal{C})$ which will accept all and only the traces σ respecting the contract expression.*

Proof: Take an expression $\mathcal{C}$ of $\mathcal{CL}$, we construct the alternating Büchi automaton $A^{\mathcal{N}}(\mathcal{C}) = (S, \Sigma, s_0, \rho, F)$ as follows. The alphabet $\Sigma = \mathcal{A}_B^{\&}$ consists of the finite set of concurrent actions. Therefore the automaton accepts traces as in Definition 1. The set of states $S = FL(\mathcal{C}) \cup \overline{FL(\mathcal{C})}$ contains the subexpressions of the start expression $\mathcal{C}$ and their negations. Note that in $\mathcal{CL}$ the negation $\neg\mathcal{C}$ is $[\mathcal{C}?]\bot$, thus $\forall \mathcal{C} \in FL(\mathcal{C})$ then $[\mathcal{C}?]\bot \in \overline{FL(\mathcal{C})}$. The initial state s_0 is the expression $\mathcal{C}$ itself.

The transition function $\rho : S \times \mathcal{A}_B^{\&} \to \mathcal{B}^+(S)$ is defined in Table 4 (the dynamic logic operators are omitted; see [14]) and is based on the following *dualizing construction*: for a Boolean formula $\theta \in \mathcal{B}^+(S)$ the dual $\overline{\theta}$ is obtained by switching $\vee$ and $\wedge$, **true** and **false**; and the dual of a state $\overline{\mathcal{C}}$ is the state $[\mathcal{C}?]\,\bot$ containing the negation of the expression. By looking at the definition of

$$\rho(\bot,\gamma) \triangleq \textbf{false} \qquad \rho(\top,\gamma) \triangleq \textbf{true} \qquad \rho(P(\alpha),\gamma) \triangleq \textbf{true}$$
$$\rho(O_{\mathcal{C}}(\alpha_{\&}),\gamma) \triangleq \text{ if } \alpha_{\&} \subseteq \gamma \text{ then } \textbf{true} \text{ else } \mathcal{C}$$
$$\rho(O_{\mathcal{C}}(\alpha \cdot \alpha'),\gamma) \triangleq \rho(O_{\mathcal{C}}(\alpha),\gamma) \wedge \rho([\alpha]O_{\mathcal{C}}(\alpha'),\gamma)$$
$$\rho(O_{\mathcal{C}}(\alpha + \alpha'),\gamma) \triangleq \rho(O_{\bot}(\alpha),\gamma) \vee \rho(O_{\bot}(\alpha'),\gamma) \vee \mathcal{C}$$
$$\rho(F_{\mathcal{C}}(\alpha_{\&}),\gamma) \triangleq \text{ if } \alpha_{\&} \not\subseteq \gamma \text{ then } \textbf{true} \text{ else } \mathcal{C}$$
$$\rho(F_{\mathcal{C}}(\alpha \cdot \alpha'),\gamma) \triangleq \rho(F_{\bot}(\alpha),\gamma) \vee F_{\mathcal{C}}(\alpha')$$
$$\rho(F_{\mathcal{C}}(\alpha + \alpha'),\gamma) \triangleq \rho(F_{\mathcal{C}}(\alpha),\gamma) \wedge \rho(F_{\mathcal{C}}(\alpha'),\gamma)$$
$$\rho([\alpha_{\&}]\mathcal{C},\gamma) \triangleq \text{ if } \alpha_{\&} \subseteq \gamma \text{ then } \mathcal{C} \text{ else } \textbf{true}$$
$$\rho([\mathcal{C}_1?]\mathcal{C}_2,\gamma) \triangleq \overline{\rho(\mathcal{C}_1,\gamma)} \vee (\rho(\mathcal{C}_1,\gamma) \wedge \rho(\mathcal{C}_2,\gamma))$$

Table 4. Transition Function of Alternating Büchi Automaton for $\mathcal{CL}$.

ρ we see that the expression $[\beta^*]\mathcal{C}$ is the only expression which requires repeated evaluation of itself at a later point (causing the infinite unwinding) in the run tree. It is easy to see that if a run tree has an infinite path then this path goes infinitely often through a state of the form $[\beta^*]\mathcal{C}$, therefore the set of final states F contains all the expressions of the type $[\beta^*]\mathcal{C}$.

The rest of the proof shows the correctness of the automaton construction.

Soundness: given an accepting run tree $(T,\mathcal{V})$ of $A^{\mathcal{N}}(\mathcal{C})$ over a trace σ we prove that $\forall x \in T$ a node of the run tree with depth $|x| = i,\ i \geq 0$, labeled by $\mathcal{V}(x) = \mathcal{C}_x$ a state of the automaton represented by a subexpression $\mathcal{C}_x \in FL(\mathcal{C})$, it is the case that $\sigma(i..) \models \mathcal{C}_x$. Thus we have as a special case that also $\sigma(0..) \models \mathcal{V}(\varepsilon) = \mathcal{C}$, which means that if the automaton $A^{\mathcal{N}}(\mathcal{C})$ accepts a trace σ then the trace respects the initial contract $\mathcal{C}$. We use induction on the structure of the expression $\mathcal{C}_x$.

Completeness: given a trace σ s.t. $\sigma \models \mathcal{C}$ we prove that the constructed automaton $A^{\mathcal{N}}(\mathcal{C})$ accepts σ (i.e. there exists an accepting run tree $(T,\mathcal{V})$ over the trace σ). □

Example 1 as alternating automata: We shall now briefly show how for the $\mathcal{CL}$ expression $\mathcal{C} = [e]O_{O_{\bot}(p \cdot p)}(p + d\&n)$ of page 3 we construct an alternating automaton which accepts all the traces that satisfy $\mathcal{C}$ and none others. The Fischer-Ladner closure of $\mathcal{C}$ generates the following set of subexpressions:

$$FL(\mathcal{C}) = \{\mathcal{C}, O_{O_{\bot}(p \cdot p)}(p + d\&n), O_{\bot}(p), \bot, O_{\bot}(d\&n), O_{\bot}(p \cdot p), [p]O_{\bot}(p)\}$$

The set $\mathcal{A}_B^{\&}$ of concurrent actions is the set $\{e,p,n,d\}^{\&}$ of basic actions closed under the constructor &. The alternating automaton is:

$$A^{\mathcal{N}}(\mathcal{C}) = (FL(\mathcal{C}) \cup \overline{FL(\mathcal{C})}, \{e,p,n,d\}^{\&}, \mathcal{C}, \rho, \emptyset)$$

Note that there is no expression of the form $[\beta^*]\mathcal{C}$ in FL because we have no recursion in our original contract clause from Example 1. This means that the automaton is accepting all run trees which end in a state where the transition

function returns **true** on the input symbol.[4] The transition function ρ is defined in table below where $\mathcal{C}_1 = O_{O_\perp(p\cdot p)}(p + d\&n)$:

$\rho(state, action)$	e	p	d	$e\&d$	$e\&p$	$d\&n$	$e\&d\&n$
$\mathcal{C}$	$\mathcal{C}_1$	**true**	**true**	$\mathcal{C}_1$	$\mathcal{C}_1$	**true**	$\mathcal{C}_1$
$\mathcal{C}_1$	$O_\perp(p\cdot p)$	**true**	$O_\perp(p\cdot p)$	$O_\perp(p\cdot p)$	**true**	**true**	**true**
$O_\perp(p)$	$\perp$	**true**	$\perp$	$\perp$	**true**	$\perp$	$\perp$
$O_\perp(d\&n)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	**true**	**true**
$O_\perp(p\cdot p)$	$\perp$	$O_\perp(p)$	$\perp$	$\perp$	$O_\perp(p)$	$\perp$	$\perp$
$[p]O_\perp(p)$	**true**	$O_\perp(p)$	**true**	**true**	$O_\perp(p)$	**true**	**true**

Computing the values in the table above is routine; e.g.:

$$\rho(\mathcal{C}_1, e) = \rho(O_\perp(p), e) \vee \rho(O_\perp(d\&n), e) \vee O_\perp(p\cdot p) = \perp \vee \perp \vee O_\perp(p\cdot p)$$

Because from the state $\perp$ nothing can be accepted (as it generates only **false**) we have written in the table only $O_\perp(p\cdot p)$. There are 2^4 labels in the alphabet of $A^{\mathcal{N}}(\mathcal{C})$ but we show only some of the more interesting ones. Moreover, none of the states from $\overline{FL}$ (i.e. $[\mathcal{C}_1?]\ \perp$, the complemented expressions) are reachable nor do they contribute to the computation of any transition to a reachable state (like e.g. $O_\perp(d\&n)$ contributes to the computation of $\rho(\mathcal{C}_1, e)$), so we have not included them in the table. The line for state $\perp$ is omitted.

4 Monitoring $\mathcal{CL}$ Specifications of Contracts

We use the method of [1] and we consequently use a 3-valued semantics approach to run-time monitoring. The monitor will generate a sequence of observations, denoted $[\sigma \models \mathcal{C}]$, for a finite trace σ defined as:

$$[\sigma \models \mathcal{C}] = \begin{cases} \mathbf{tt} & \text{if } \forall \sigma' \in \Sigma^\omega : \sigma\sigma' \models \mathcal{C} \\ \mathbf{ff} & \text{if } \forall \sigma' \in \Sigma^\omega : \sigma\sigma' \not\models \mathcal{C} \\ ? & \text{otherwise} \end{cases}$$

We use a standard method [16] to construct a exponentially larger nondeterministic Büchi automaton $NBA(\mathcal{C})$ from our alternating automaton $A^{\mathcal{N}}(\mathcal{C})$ s.t. both automata accept the same trace language. Therefore $NBA(\mathcal{C})$ is exponential in the size of the expression.

The method of [1] is the following: take the $NBA(\mathcal{C})$ for which we know that $[\sigma \models \mathcal{C}] \neq \mathbf{ff}$ if there exists a state reachable by reading σ and from where the language accepted by $NBA(\mathcal{C})$ is not empty. Similarly for $[\sigma \models \mathcal{C}] \neq \mathbf{tt}$ when taking the complement of $NBA(\mathcal{C})$ (or equivalently we can take the $NBA(\neg\mathcal{C})$ of the negated formula which is $[\mathcal{C}?]\ \perp$). Construct a function $F : S \to \{\top, \perp\}$ which for each state s of the $NBA(\mathcal{C})$ returns $\top$ iff $\mathcal{L}(NBA(\mathcal{C}), s) \neq \emptyset$ (i.e. the language accepted by $NBA(\mathcal{C})$ from state s is not empty), and $\perp$ otherwise. Using F one can construct a nondeterministic finite automaton $NFA(\mathcal{C})$ accepting finite

[4] Note that for this particular example we do not see the power of alternating automata. More, the alternating Büchi automata behaves like a NFA.

traces s.t. $\sigma \in \mathcal{L}(NFA(\mathcal{C}))$ iff $[\sigma \models \mathcal{C}] \neq \bot$. This is the same NBA only that the set of final states contains all the states mapped by F to $\top$. Similarly construct a $NFA(\neg\mathcal{C})$ from $NBA(\neg\mathcal{C})$. One uses classical techniques to determinize the two NFAs. Using the two obtained DFAs one constructs the monitor as a finite state Moore machine which at each state outputs $\{\mathbf{tt}, \mathbf{ff}, ?\}$ if the input read until that state respectively satisfies the contract clause $\mathcal{C}$, violates it, or it cannot be decided. The monitor is the product of the two $DFA(\mathcal{C})$ and $DFA(\neg\mathcal{C})$.

We need that the monitor can read (and move to a new state) each possible action from the input alphabet. When doing the product of the two DFAs, if one of them does not have a transition for one of the symbols then this is lost for the monitor too. Therefore we add to each DFA a dummy state which is not accepting and which collects all the missing transitions for all states.

Correctness of the method [1] states $\lambda(\rho(s_0, \sigma)) = [\sigma \models \mathcal{C}]$, i.e. the output function $\lambda : S \to \{\mathbf{tt}, \mathbf{ff}, ?\}$ of the Moore machine returns for the state reached by reading σ from the starting state s_0 the semantics of $\mathcal{C}$ on the finite trace σ. The monitor generated is proven to have size double-exponential in the size of the expression; one exponent coming from translation of $\mathcal{A}^{\mathcal{N}}$ into the NBA and the other from determinization.

5 Conclusion

The work reported here may be viewed from different angles. On one hand we use *alternating automata* which has recently gained popularity [8] in the temporal logics community. We apply these to our rather unconventional logic $\mathcal{CL}$ [12]; i.e. a process logic (PDL [4]) extended with deontic logic modalities [18]. On another hand we presented the formal language $\mathcal{CL}$ with a trace semantics, and showed how we specify electronic contracts using it. From a practical point of view we presented here a first fully automated method of extracting a run-time monitor for a contract formally specified using the $\mathcal{CL}$ logic.

Note that our main objective is not to enforce a contract, but only to monitor it, that is to observe that the contract is indeed satisfied. The trace semantics presented in this paper is intended for monitoring purposes, and not to explain the language $\mathcal{CL}$. Thus, from the trace semantics point of view $[\alpha_{\&}]\mathcal{C}$ is equivalent to $F_{\mathcal{C}}(\alpha_{\&})$, but we need such a distinction since this is not the case in $\mathcal{CL}$ (see $\mathcal{CL}$ branching semantics [13]).

Related work: For run-time verification our use of alternating automata on infinite traces of actions is a rather new approach. This is combined with the method of [1] that uses a three value (i.e. *true*, *false*, *inconclusive*) semantics view for run-time monitoring of LTL specifications. We know of the following two works that use alternating automata for run-time monitoring: in [3] LTL on infinite traces is used for specifications and alternating Büchi automata are constructed for LTL to recognize *finite* traces. The paper presents several algorithms which work on alternating automata to check for word inclusion. In [15] LTL has semantics on *finite* traces and nondeterministic alternating finite automata are used to recog-

nize these traces. A determinization algorithm for alternating automata is given which can be extended to our alternating Büchi automata.

We have taken the approach of giving semantics to $\mathcal{CL}$ on *infinite* traces of actions which is more close to [3] but we want a deterministic finite state machine which at each state checks the finite input trace and outputs an answer telling if the contract has been violated. For this reason we fount the method of [1] most appealing. On the other hand a close look at the semantics of $\mathcal{CL}$ from Section 2 reveals the nice feature of this semantics which behaves the same for finite traces as for infinite traces. This coupled with the definition of alternating automata from Section 3 which accepts both infinite and finite traces gives the opportunity to investigate the use of alternating finite automata from [15] on the finite trace semantics. This may generate a monitor which is only single-exponential in size.

References

1. A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS'06*, volume 4337 of *LNCS*, pages 260–272. Springer, 2006.
2. A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
3. B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.
4. M. J. Fischer and R. E. Ladner. Propositional modal logic of programs. In *9th ACM Symposium on Theory of Computing (STOC'77)*, pages 286–294. ACM, 1977.
5. S. Göller, M. Lohrey, and C. Lutz. PDL with Intersection and Converse Is 2 EXP-Complete. In *FoSSaCS*, volume 4423 of *LNCS*, pages 198–212. Springer, 2007.
6. D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic*. MIT Press, 2000.
7. D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.
8. O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of ACM*, 47(2):312–360, 2000.
9. Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In *POPL'87*, pages 1–12, 1987.
10. D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *LICS*, pages 422–427. IEEE, 1988.
11. V. R. Pratt. Process logic. In *POPL'79*, pages 93–100. ACM, 1979.
12. C. Prisacariu and G. Schneider. A formal language for electronic contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.
13. C. Prisacariu and G. Schneider. CL: A Logic for Reasoning about Legal Contracts – Semantics. Technical Report 371, Univ. Oslo, 2008.
14. C. Prisacariu and G. Schneider. Run-time Monitoring of Electronic Contracts – theoretical results. Technical report, Univ. Oslo, 2008.
15. V. Stolz and E. Bodden. Temporal Assertions Using AspectJ. In *RV'05*, volume 144 of *ENTCS*, pages 109–124. Elsevier, 2006.
16. M. Y. Vardi. Alternating Automata: Unifying Truth and Validity Checking for Temporal Logics. In *CADE*, volume 1249 of *LNCS*, pages 191–206. Springer, 1997.
17. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE, 1986.
18. G. H. von Wright. Deontic logic. *Mind*, 60:1–15, 1951.