Simulation Semantics for formal e-contracts

Luis Llana¹, María-Emilia Cambronero², and Gregorio Díaz²
llana@dfi.uclm.es, memilia.cambronero@uclm.es,
gregorio.diaz@uclm.es

¹. Computer Science Dept. Complutensis University of Madrid
². Computer Science Dept. University of Castilla-La Mancha

Abstract. Relationships between entities in today’s increasingly interconnected context have grown in complexity and evolved from simple communication processes to more complicated distributed systems. It is in this context that we aim to develop a consistent definition for these relationships together with a set of techniques to check their proper use. In this paper we present a process algebra to describe these contract relationships and a set of formal machinery to determine whether an implementation follows the rules established by these contracts. The main formal technique used is a simulation relation where an implementation is checked step by step against a given contract. Several toy examples are provided to facilitate understanding of the formal definitions.

1 Introduction

Nowadays many transactions are conducted electronically, for both individuals and companies. For instance, when a bank customer withdraws money from an automatic teller machine or when a shop allows clients to browse, select, buy and pay its products on the Internet, an electronic transaction takes place. Normally, the parties involved in the transaction never physically meet each other. In this context, some rules are defined in order to guarantee these transactions, specifically in the software context, where the term contract has traditionally been used as a metaphor to represent types of agreements between software elements at different levels of abstraction, as described in [5]. These contracts are called electronic contracts, e-contracts for short.

A well-known formalism to specify contracts is deontic logic [20], which is concerned (among other things) with the formalization of contracts, specifically with moral and legal obligations, permissions, and prohibitions, their interrelation and properties, as well as events/consequences resulting from violations of obligations and prohibitions. Therefore, it is important to take the time to analyze whether these contracts, in the form of permissions, obligations and prohibitions, are fulfilled by the different parties involved in the contract. In this paper, we present a formalism to analyze e-contracts based on timed process algebras [7,18,10]. But in our case, the underlying semantics is that of timed simulation [19].
Simulation is an important in many areas of computer science (model checking, concurrency theory, formal verification...). It is both a theoretical (e.g. [1,7,?,13]) and practical (e.g. [7,17]) area of research. The use of practical implementation applications for e-contracts is particularly interesting its use in model checking minimization ([3,11]) as a technique to overcome the state explosion problem. Then one of the main aims of this paper is to use simulation semantics to model and analyze electronic contracts (e-contracts) taking advantage of this model checking minimization.

Therefore, this paper defines a process algebra to describe e-contracts relationships and a set of formal techniques to determine whether an implementation follows the rules established by these e-contracts. In particular we present a formal semantics for e-contracts in order to capture the deontic normative concepts of permission, obligation and prohibition, as well as the penalties in cases of certain violations. In our formal framework each party involved in a contract is represented by a contract agent. We have defined a simulation preorder between agents. This preorder can be used as an implementation preorder: if the contract agent $P$ simulates the contract agent $Q$, then $Q$ can be replaced by $P$ in any contract.

Deontic logic was first proposed in [20], since then several works have proposed its use for reasoning about contracts. Prisacariu and Schneider in [15] present a formal language for e-contract, which is based on deontic notions of permission, obligation and prohibition. They propose an extension of the $\pi$ – calculus in order to express concurrent actions and to capture the meaning of the deontic notions, but their contract language lacks the possibility to express time constraints and they propose adding real-time as an extension to express and reason about contracts with deadlines. Governatory et al. [7] present a formalism for the representation of contracts also using deontic logic, by including the representation and reasoning about violations of obligations in contracts. They also provide a mechanism to check whether business processes comply with business contracts, using the FCL logic (based on deontic logic) to reason about contracts. In [12] C. Prisacariu et al. shows how to obtain a run-time monitor for contracts written in Contract Language (CL [16]), which is an action-based formal language tailored for writing e-contracts and which allows to write conditional obligations, permissions, and prohibitions to be written over actions. They also show the way to specify reparations. The trace semantics of (CL) formalizes the notion of a trace fulfilling a contract. The authors use alternating Büchi automaton and they claim that it accepts the traces (and only 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 CL contracts. Their focus is on verification or monitoring. Our approach is based on simulation semantics. Previous works [9,8] show that simulation can be a useful technique to reduce the number of states of a model. They utilize a variant of simulation that can be used as a conformance relation, with the aim of extracting tests from a specification. We propose to translate these results to our framework. Our work also considers timed restrictions in the system. Other works do not use deontic
logic to describe contracts. For example, in [7] contracts are formally defined
as a set of commitments and they reason about the correctness of the contract
specification.

In this paper we have chosen a visual model for the design of contracts [5]
since it implements many of the desirable properties of a good formal language
for normative texts, as presented in [14]. This model is composed of Contract-
Oriented Diagrams (C-O Diagrams), which allow the representation of complex
clauses describing the obligations, permissions, and prohibitions of different sig-
natories, as well as reparations. Also, C-O Diagrams permit users to define real-
time constraints. We use these diagrams only for representation issues, since
a graphical representation helps to more easily understand complex contracts,
which can be composed of many clauses.

The paper is structured as follows: Section 2 presents an overview of the
visual model for e-contracts. Next, in Section 3, we present the syntax and the
operational semantics of our formalism. Section 5 develops the formal semantics
of language. Section 6 presents a case study of a Coffee Machine. Finally, the
conclusions and future work are described in Section 7.

2 The visual model for e-contracts: C-O diagrams

C-O diagrams presented in [5] are a simplified visual model to represent e-
contracts. We will use this model to facilitate the task of specifying contracts.
The diagram type used here differs slightly from the one presented in [5] to
adapt it to the specific goals of this work. Figure 1 depicts a typical example of
a coffee machine, which consists of two agents (the machine and a customer).
This example is explained in detail in the case study section, but first we will
present the elements these diagrams consists of, so readers can start to familiar-
ize themselves with them. The main difference is the use of variables, which are
not considered in this first stage of this proposal.

Fig. 1. C-O Diagram for Coffee Machine specification.
The diagram depicts a set of boxes and arrows to interconnect boxes. A box, known as “clause”, can be divided in four different cells. The central cell, the “proposition”, specifies the obligations, prohibitions, permissions, actions and refinements. The left-hand cells define the upper (top) and lower (bottom) bounds defining the interval in which the clause must be enacted. In some cases, it might not be necessary to declare bounds and $[0, \infty]$ is taken as default interval. The last cell on the right-hand side of the clause is the recuperation cell, which is only declared for obligations and prohibitions. This cell declares a pointer to another clause, which is activated whenever the behaviour declared in the central cell, the proposition, is violated. An extra element is added to the clauses at the top, where two agents are defined. The first agent on the left performs the action and the agent on the right is the receiver of the action. These elements are only used and mandatory if the proposition specifies one or more actions.

An Arrow connecting clauses specifies a sequence, but if it targets a predecessor clause it defines a recursion refinement. If an arrow targets a clause with either OR or AND, then this targeted clause must connect with two other clauses behaving as the disjunctive or conjunctive refinements.

As in process algebra, any communication involves two agents: one that emits the communication and another one that receives. This is similar in our case. For instance in the case of an obligation there is one or more agents that have the obligation to perform an action and another agent receives. The agent that is supposed to receive an action can detect that the contract is not valid if the action is not received. For instance let us suppose that we have a simple contract where agent $P$ has the obligation to perform an action within the time interval $[2, 8]$, the action is received by agent $Q$. In our formalism these agents can be specified by:

\[
\text{Example 1. } P \xrightarrow{a?} Q, \quad P = a! [2, 8], \quad Q = O_b(a?)[2, 8]
\]

Permissions allow agents to receive actions from other agents in a specific time interval. For instance, Example 2 shows an agent $P$ allowing other agent $Q$ to perform the action $a$ in the interval $[3, 5]$. An implementation of $Q$, where the action is performed during this implementation will be valid. On the contrary, if an implementation specifies a longer or shorter interval for instance $[4, 7]$ or $[1, 4]$, respectively, then it might lead to the contract violation and, therefore to an invalid implementation. In this case, $P_r(a?)[3, 5]$ in $P$ grants the other process the permission of execute the input $a!$.

\[
\text{Example 2. } P \xrightarrow{a?} Q, \quad P = P_r(a?)[3, 5], \quad Q = a! [3, 5]
\]

Prohibitions are the most complex deontic operator since they combine the idea of not allowing a certain set of actions during a given interval of time. Example 3 specifies a e-contract where action $a$ is forbidden during the interval

\[1\] For these two types of propositions when the main proposition fails and no reparation is defined the contract is violated.
[3, 5] in the behavior of an AND refinement where the left clause permits action a during [1, 5] while on the right part action b must be performed in [1, 4]. Action a is only permitted in the interval [1, 2] by agent Q as a result of the interaction between the prohibition and the permission as it can be observed in the specification of process Q. An implementation performing a outside this interval leads to a violation of this contract.

Example 3.

\[
\begin{array}{c|c|c|c}
Q & P & O(b?) & Q \\
\hline
3 & F(a?) & 4 & Q \\
\hline
1 & P(a?) & 5 & Q \\
\end{array}
\]

\[ P = F_b(\{a?\})[3, 5] \text{ in } (P_r(a?))[1, 5] \mid \]
\[ O(b?)[1, 4] \]
\[ Q = a![1, 2] \mid b![1, 4] \]

\[ \square \]

3 Syntax

In this section we first establish the e-contract model, which captures the main deontic aspects of e-contracts that have been briefly described in the previous section, after which we define an operational semantics for it. For our purposes, a specification is described by Definition 1. In this definition we are considering a set of inputs I (a?, b?,... will denote individual inputs), a set of outputs O (a!, b!... will denote outputs). Each input and output has a counterpart in the other set: a? ∈ I iff a! ∈ O. For any a ∈ I ∪ O, we define its counterpart as a. We need two special actions vc, ic / I ∪ O to signal the correct finalization of a contract and an invalid contract, the set of actions Act = I ∪ O ∪ {ic, vc} (a, b,... denote individual actions), a silent action τ / Act, the set Actr = Act ∪ {τ}, and a set of agent variables Var (x, y, z will denote individual agent variables). We will also need a discrete time domain T (t, t1, t2... will denote time units), we will assume that there is a minimum time unit δ and all the time units are multiple of that minimum time unit, we need a variation of the subtraction operation: t1 − t2 = max(0, t1 − t2). We will need the infinity, ∞ / T, to represent a behavior that is always available; ∞ / t = ∞.

Definition 1. The set of contract agent terms is defined by the following BNF:

\[ P ::= P_r(a?)[t_1, t_2] \mid O_b(a?)[t_1, t_2] \triangleright R \mid F_b(A)[t_1, t_2] \text{ in } P \triangleright R \mid a![t_1, t_2] \mid \]
\[ \text{ ic } \mid \text{ vc } \mid \text{ rec } x.P \mid x \mid P_1; P_2 \mid P_1 \sqcup P_2 \mid P_1 \sqcap P_2 \mid P_1 \parallel P_2 \]

Where P_r, O_b and F_b are the deontic operators for the permissions, obligations and prohibitions, respectively. a? ∈ I, a! ∈ O, t_1 ∈ T, t_2 ∈ T ∪ {∞}, A ⊆ I, P, R, P_1, P_2 are agent terms, and x ∈ Var. We will consider the set of contract agents (or simply agents), written AG, as the set of closed contract agents terms.

We define the language of P ∈ AG, written L(P), as the set of inputs and outputs appearing in P.

\[ \square \]

Therefore, P_r(a?)[t_1, t_2] models the permission to perform action a in a time interval [t_1, t_2]. O_b(a?)[t_1, t_2] ∨ R represents the obligation to perform action a in a time interval [t_1, t_2]; if the obligation is not fulfilled it is necessary to follow
the behavior specified by the reparation $R$. $F_b(a)[t_1, t_2]$ in $P \triangleright R$ depicts the prohibition of performing an input action subset ($A$) in a time interval $[t_1, t_2]$, in case this prohibition is violated a reparation specified by $R$ must be performed. $a[t_1, t_2]$ can execute the output action at any time in the time interval $[t_1, t_2]$. This operator represents an agent that can perform the input at any time in the interval, so the choice of the time is an internal choice of the agent. $vc$ defines a valid contract, while $ic$ defines an invalid contract. $x \in Var$, where $Var$ defines a set of agent variables, is used to define the recursion operator $rec \times P$. This operator models the recursion of $P$, that is, the contract repetition. $P_1; P_2$ represents the concatenation in sequence. $P_1 \sqcap P_2$ defines a deterministic choice (or external choice), where the environment has the possibility to make the choice between $P_1$ and $P_2$. $P_1 \sqcap P_2$ defines a nondeterministic choice (also called internal choice), where the system is taking the decision and the environment has no control over it. $P_1 || P_2$ models the parallel execution of two agents, $P_1$ and $P_2$.

In our examples in the paper we will use some shortcuts: If the time interval is omitted (for instance $P_r(a?)$), the time interval is $[0, \infty]$ ($P_r(a?)\{0, \infty\}$). If the reparation is omitted (for instance $O_b(a?)$), we assume that is $ic$ ($O(b(a?) \triangleright ic)$).

4 Operational Semantics

Figures 2 and 3 show the operational semantics. There are two kinds of transitions: timed transitions and action transitions. Timed transitions expresses the time delay while action transitions expresses the execution of an action. The base operator $vc$ can delay time (rule $vc2$) and signals the correct termination of a contract (rule $vc1$). Similarly, the operator $ic$ can delay time (rule $ic1$) and signals an incorrect contract (rule $ic2$). Rule $rec1$ indicates the unfolding of the recursion operator. This unfolding is urgent, so must be done immediately (rule $rec2$).

The input action, the permission operator and the obligation operator can only delay time until the lower bound of the interval is reached (rules $act1$, $perm1$, $obl1$). When an output action is enabled, the operator chooses internally the time when the action will be executed (rule $act2$), then the time passes until that time. When this time is reached and the action is executed the computation proceeds normally (rule $act3$), otherwise if the action cannot be executed (because there are no other agents willing to synchronized) the invalid contract signal is raised (rule $act4$).

The permission and obligation operator behaves in the same way until the upper bound of the interval is reached: while the input action is enabled, both operators can execute the action (rules $perm3$ and $obl3$) and both operators can delay time until the upper bound of the interval (rules $perm2$ and $obl2$). The main difference between $perm2$ and rule $obl2$ comes from their behaviors when time overtakes the upper bound of the interval. In the case of the permission the computation evolves normally, while in the case of the obligation the part of the recuperation is enabled.
Fig. 2. Basic operator rules
The forbidden operator behaves normally as long as the time interval is not enabled (rules forb1 and forb4). When time reaches the upper bound of the interval, the operator is disabled (rule forb2). While the time interval is enabled only actions not belonging to the set of forbidden actions can be performed (rule forb5) and if a forbidden action is performed, the part of the recuperation is enabled (rule forb3).

The rules in Figure 3 deal with the compound operators. They are the typical rules of a timed process algebra [18]. The internal choice operator can only make the internal decision to behave as one of its components (rules icho1 and icho2), rule icho3 is necessary for Proposition 2. The external choice can let time pass if both components can (rule cho1): when one of the components of the choice is able to execute an action then the other component is disabled (rules cho2 and cho3), while the execution of internal actions does not disable the choice (rules cho4 and cho5).

The timed rule of the parallel operator is the most complex of the timed rules (rule par1). There are two obvious conditions for the parallel composition being able to let time pass: both components can let time pass. But if at some intermediate point a synchronization or termination is available, it must be executed before letting time pass. The synchronization of two components is translated by a silent move (rule par4). In order to terminate, both components of the
operator must terminate (rule par5). Finally any of the components can evolve autonomously (rules par2 and par3).

The rules of the sequence are simple. The first component can execute actions or let time passes until it terminates (rules seq1 and seq3). We consider termination urgent: it is executed as soon it is available; so we cannot let time pass in rule seq3 if the first component has terminated. When this happens, the control passes to the second component (rule seq2).

First we will prove some basic properties of the operational semantics. It is important to note that some rules have negative premises that could lead to an inconsistent semantic (for instance $P \rightarrow Q \iff P \nrightarrow Q$).

**Proposition 1.** The operational semantics is consistent

*Proof.* This result follows the fact that the action transitions does not depend on the time transitions and there are not negative transitions to define them, so its corresponding semantic is consistent. Once the action transitions are defined, the rules for the timed transitions have only negative rules on the action transitions, so the semantics derived for the timed transitions is also consistent. \qed

Other propositions that the semantic of the timed transitions verifies are the following basic properties:

**Proposition 2.** Let $P, P', P'' \in \mathcal{AG}, t, t' \in T$, then the following properties:

- $P \sim P$. 
- If $P \sim P'$ and $P \sim P''$, then $P' = P''$. 
- If $P \sim P$. Then for any $0 < t' < t$ there exists $P''$ such that $P \sim P''$. 
- If $P \sim P', P \sim P''$, then $P \sim P''$.

*Proof.* It is done by simple structural induction. \qed

The associative and commutative properties of the binary operators help to avoid writing excessive parenthesis in the terms.

**Proposition 3.** The operators $\Box$, $\cap$, and $|||$ are commutative and associative.

*Proof.* The set of derived transitions are the same in terms with those operators regardless the order of the members or the order of the parenthesis. \qed

Next we can define the notion of contract, valid contract and valid implementation. First we have the notion of contract, which is the parallel composition of several agents. Since the parallel operator is associative and commutative, the order among the agents is not relevant and the parenthesis are not necessary.

**Definition 2.** Let $n \in \mathbb{N}$ be a natural number and let $P_i$ be a contract agent for $1 \leq i \leq n$. A contract is specified by the parallel composition of the agents: $C = P_1 || P_2 || \cdots || P_n$. \qed

In order to define the notion of valid contract we need the following notation to simplify the definition.
Definition 3. Let \( P, P' \) be contract agents, \( a \in \text{Act}_r \) and \( t \in T \), we write \( P \xrightarrow{t,a} P' \) iff there is a contract agent \( P'' \) such that \( P \xrightarrow{\sim} P'' \xrightarrow{t,a} P' \). \( \Box \)

So a valid contract is one that when executed never yields the signal \( ic \) of invalid contract.

Definition 4. Let \( C \) be a contract, an interaction of the contract is a sequence of contracts \( C_0 = C, C_1, \ldots, C_n \) such that there are transitions

\[
C_0 \xrightarrow{t_0,\tau} C_1 \xrightarrow{t_1,\tau} \cdots \xrightarrow{t_{n-2},\tau} C_{n-1} \xrightarrow{t_{n-1},\tau} C_n
\]

A contract is invalid if there is a interaction of the contract \( C_0 = C, C_1, \ldots, C_n \) such that \( C_n \xrightarrow{ic} \). A contract is valid if it is not invalid. \( \Box \)

Definition 5. Let \( C = A_1 \parallel A_2 \parallel \cdots \parallel A_k \parallel \cdots \parallel A_n \), we say that an implementation \( I \) is correct for agent \( A_k \) iff the contract \( C' \) obtained by substituting the agent \( A_k \) by the implementation \( I \), \( (C' = A_1 \parallel A_2 \parallel \cdots \parallel I \parallel \cdots \parallel A_n) \), is correct. \( \Box \)

5 Simulation semantics

In this section we are going to provide and discuss a simulation semantics for the agents of a contract. The semantics defined in the previous section (Definition 5) is difficult to check. We believe that simulation semantics is a good alternative. It can be computed efficiently [6] and can be used to reduce the number of states as a previous step in other formal techniques such as model checking.

Before the definition of the simulation semantics we need some notation.

Definition 6. Let \( P \in \mathcal{AG}, t \in T \) and \( a \in I \cup O \), we define the transitions:

- \( P \xrightarrow{t} P' \) iff there are \( P_i \in \mathcal{AG} \) \((0 \leq i \leq n + 1)\) and \( t_i \in T \) \((0 \leq i \leq n)\) such that: \( P = P_0 \xrightarrow{t_0} P_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} P_n \xrightarrow{\tau} P_{n+1} = P' \) where \( t = \sum_{i=0}^{n} t_i \).
- \( P \xrightarrow{t,a} P' \) iff there are \( P_i \in \mathcal{AG} \) \((0 \leq i \leq n + 1)\) and \( t_i \in T \) \((0 \leq i \leq n)\) such that: \( P = P_0 \xrightarrow{t_0} P_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} P_{n-1} \xrightarrow{t_n,a} P_n \xrightarrow{\tau} P_{n+1} = P' \) where \( t = \sum_{i=0}^{n} t_i \).
- \( \text{Act}_r(P) = \{(a,t) \mid \exists P' \in \mathcal{AG}, a \in I \cup O \cup \{e\} : P \xrightarrow{t,a} P'\} \)

Next we define our concept of simulation. It is a timed simulation semantics and follows the classical co-inductive schema. The only particularity is that we want to restrict the simulation to the set of actions specified in the contract; since the implementation could perform actions not specified in the contract. \( ^2 \) Let us recall that \( L(C) \) is the set of inputs and outputs appearing in \( C \) (Definition 1).

Definition 7. Let \( S \) be a relation of agents \((S \subseteq \mathcal{AG} \times \mathcal{AG})\). \( S \) is a simulation relation for a contract \( C \) iff whenever \((P, Q) \in S\) the following conditions hold:

\(^2\) There are not restrictions for the execution of those actions.
\[ - \text{Act}(Q) \cap L(C) \times T = \text{Act}(P) \cap L(C) \times T. \]
\[ - \text{For any } (a, t) \in \text{Act}_t(Q), \text{ and } t \in T, \text{ if } P \xrightarrow{a, t} P', \text{ then there exists } Q' \text{ such that } Q \xrightarrow{a, t} Q' \text{ and } (P', Q') \in S. \]

**Definition 8.** Let \( i, s \in AG \) two agents for a contract \( C \), we say that \( P \) simulates \( Q \), written \( P \preceq Q \), iff there exists a simulation \( S \) for the contract \( C \) such that \((P, Q) \in S.\]

The simulation we have defined is a preorder so it can be used as an implementation relation.

**Proposition 4.** The relation \( \preceq \) for a contract \( C \) is a reflexive and transitive relation.

**Proof.** This can be checked easily:
\[ - \text{id} = \{(P, P) \mid P \in AG\} \text{ is a simulation.} \]
\[ - \text{If } R \text{ and } S \text{ are simulations, the composition relation } R \circ S \text{ is a simulation.} \]

Next we present the main result in this section. The simulation relation can be used to obtain a valid implementation of an agent.

**Theorem 1.** Let \( C = AG_1 || AG_2 || \cdots || AG_k || \cdots || AG_n \) a valid contract. If \( P \preceq AG_k \), then \( P \) is a valid implementation of \( AG_k \).

**Proof.** Let us suppose that \( i \) is not a valid implementation. There is an interaction evolution of the contract \( C' \) resulting from replacing the agent \( AG_k \) with the implementation \( P \) that yields the invalid contract signal:
\[ C' = C_0 \xrightarrow{t_1} C_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} C_n \text{ with } C_n \not\rightarrow \]

That interaction can be unzipped into an evolution of the implementation and the rest of contract agents. Let us consider the evolution of the implementation:
\[ P = P_0 \xrightarrow{t_1} P_1 \xrightarrow{t_2} P_2 \cdots \xrightarrow{t_n} P_n \]

Since in the evolution interaction of the contract, the actions that are not in the contract are prohibited, we obtain that \( a_i \in L(C) \ (1 \leq i \leq n) \). Since \( P \preceq AG_k \) there is an evolution of the agent as follows
\[ AG_k = Q_0 \xrightarrow{t_1} Q_1 \xrightarrow{t_2} Q_2 \cdots \xrightarrow{t_n} Q_n \]

If this computation can be zipped with the computation of the rest of the agents, then we have found that the contract is invalid. So that computation cannot be zipped with the rest of the contracts, let us consider the first transition \( Q_{j-1} \xrightarrow{t_j} Q_j \) that cannot be zipped with the rest of the agents. If this is possible, it is because this transitions can be decomposed as follows:
\[ Q_{j-1} \xrightarrow{t'_{j-1}} Q' \xrightarrow{t_j-t'_{j-1}} Q_j \]
and there is some $a \in I \cup O$ such that $Q \xrightarrow{a} P$ and that transition synchronizes with some of the agents. The computation $P_{j-1} \xrightarrow{t_j a_j} P_j$ can be decomposed as follows

$$P_{j-1} \xrightarrow{t_j a_j} P' \xrightarrow{t_j a_j} P_j \text{ and } P' \xrightarrow{b_j} P'$$

Since $P_{j-1} \preceq Q_{j-1}$ and $(a, t') \in \text{Act}_r(Q_j)$, we deduce that transition $P_{j-1} \xrightarrow{t_j a_j} P'$ cannot be of the form $P_{j-1} \xrightarrow{0} i'$. Since $P_{j-1} \preceq Q_{j-1}$ there exists $Q''$ such that $Q_{j-1} \xrightarrow{t_j a_j} Q''$ and $P'' \preceq Q''$. So $Q'' \xrightarrow{b_j} P''$. From $Q''$ we can build a new evolution of the agent

$$AG_k = Q_0 \xrightarrow{t_1 a_1} Q_1 \xrightarrow{t_2 a_2} Q_2 \cdots Q_{j-1} \xrightarrow{t_j a_j} Q_j' \cdots t_n a_n Q''_n$$

If that computation can be zipped with the rest of the agents, then we again obtain a contradiction. Otherwise we can proceed as before. We cannot iterate this infinitely many times since the transition $P_{j-1} \xrightarrow{t_j a_j} P'$ cannot be of the form $P_{j-1} \xrightarrow{0} P'$. \hfill \Box

**Example 4.** The reverse property is not true. It is enough to consider a valid contract where the agents never yields the invalid contract signal, for instance let us consider the contract agents $P_1 = P_1(a?)[0,10]$, $P_2 = a![0,10]$, and $P_3 = P_3(b?[0,10])$. Then, $i = \text{vc}$ is a valid implementation of $P_3$. \hfill \Box

### 6 Case Study

The case study presented in this section is inspired by the example described in [5,4]. It consists of “A Coffee Machine” involving the interaction between two different agents: a customer and a coffee machine. The coffee machine system starts when a customer orders a drink by inserting money and selecting a beverage. Coffee can be chosen either with or without milk. The machine proceeds to pour the selected drink, provided the money paid covers its exact price and the correct coins are used. If not, the machine refunds the inserted coins. After payment customers have 30 seconds to choose between either a coffee or a latte. The money is refunded if no option is selected in this interval. The order can be cancelled and the customer refunded in an interval of 10 seconds or the selected drink is poured. Note that a machine only accepts 10, 20 or 50 cent coins.

Figure 1 depicts the C-O Diagram for this coffee machine system and its specification, following the syntax given above for e-contracts, is given next:

**Example 5.** Coffee machine contract specification

$$\begin{align*}
\text{Machine} & := \text{rec } x.F_b\{C_{1\ell}, C_{2\ell}, C_{2e}\}\text{ in } \\
& \text{In}_x \text{Coins}_{50} ; \\
& (P_l(B_{\text{abort}?})[0,10]; \text{refund}[0,30]; x)\square \\
& (Q_c(B_{\text{coffee}?})[0,30] \triangleright (\text{refund}[0,30]; x); (P_{\text{coffee}}[10,30] \cap \text{refund}[30_{\ell}, 60]))\square \\
& (Q_c(B_{\text{latte}?})[0,30] \triangleright (\text{refund}[0,30]; x); (P_{\text{latte}}[10,30] \cap \text{refund}[30_{\ell}, 60])) ; x \\
& \triangleright (\text{refund}[0,30]; x)
\end{align*}$$
where the number $30 + \delta$ represents the number $30 + \delta$ and the $In_Coins_{50}$ agent is defined as follows:

$$In_Coins_{50} := P(C_{50}?) □ (P(C_{20}?) □ (P(C_{10}?) □ (P(C_{5}?) □ (P(C_{2c}?) □ (P(C_{1c}?) □ In_Coins_{49}) □ In_Coins_{48}) □ In_Coins_{45}) □ In_Coins_{40}) □ In_Coins_{30}) □ In_Coins_{50})$$

where the other agents $In_Coins_x$ are defined similarly. The customer is specified as follows:

$$Client := rec x. Out_Coins_{50}; \left\{ \begin{array}{l} B_{\text{abort}}[0, 10]; O_b(\text{refund}?[0, 30]) \cap \\ B_{\text{coffee}}[0, 30]; O_b(P_{\text{coffee}}?[10, 30]) \triangleright O_b(\text{refund}?[0, 30]) \cap \\ B_{\text{latte}}[0, 30]; O_b(P_{\text{latte}}?[10, 30]) \triangleright O_b(\text{refund}?[0, 30]) \end{array} \right\}: x$$

The agent $Out_Coins_{50}$ is similar to $In_Coins_x$:

$$Out_Coins_{50} := C_{50}! □ (C_{20}! □ (C_{10}! □ (C_{5}! □ (C_{2c}! □ (C_{1c}! □ Out_Coins_{49}) □ Out_Coins_{48}) □ Out_Coins_{45}) □ Out_Coins_{40}) □ Out_Coins_{30}) □ Out_Coins_{50})$$

where the other agents $Out_Coins_x$ are defined similarly.

This example 5 is a simple transcription from the C-O diagram. This e-contract model consists of the two agents described in the diagram the coffee machine and the customer. The next step in analyzing this contract is to compare a set of implementations. We check whether these implementations can play the role of one of the specified agents.

Example 6 consists of three implementations for the customer agent. In the first implementation the customer inserts a 50 cent coin, pushes the coffee button in the interval $[10, 20]$ and waits for the coffee in the interval $[10, 30]$. This specification coincides with that given for the customer, however the interval in which the customer pushes the coffee button is smaller than the one given by the specification. An implementation is correct either when outputs are produced in an interval that fully coincides with the specification or when this interval is included into the contract specified interval. The second implementation also varies with the customer specification in the interval given for the obligation of pouring coffee $[10, 20]$ instead of $[10, 30]$. Here as in the first implementation a smaller interval is defined. However now the contract behaviour is not guaranteed since, as specified in the contract, the machine might produce the coffee during the last ten time units. In the third implementation, it is clear that the contract is not fulfilled by this implementation since the latte button might be pushed after the upper bound specified in the contract.

Example 6. Client implementations

$$Imp^1_{Client} = rec x.C_{50}!; B_{\text{coffee}}[10, 20]; O_b(P_{\text{coffee}}?[10, 30] \triangleright \text{refund}[0, 30]); x$$
$$Imp^2_{Client} = rec x.C_{50}!; B_{\text{coffee}}[0, 30]; O_b(P_{\text{coffee}}?[10, 20] \triangleright \text{refund}[0, 30]); x$$
$$Imp^3_{Client} = rec x.C_{50}!; B_{\text{latte}}[20, 40]; O_b(P_{\text{latte}}?[10, 30] \triangleright \text{refund}[0, 30]); x$$
7 Conclusion and Future Work

In this paper we have presented a formalism based on process algebra to express e-contracts. We have defined an operational semantic, a notion of valid contract and the notion of implementation. Furthermore, we have defined a notion of simulation that is correct with regard to the notion of implementation. However, but the simulation relation is not complete: not all valid implementation of an agent simulates the agent. One of our future work lines is to find a relation that reduces the gap between the notion of correct implementation and the notion of simulation.

The implementation issues have not been discussed in detail in this proposal. We expect to implement the features defined in this paper by using the mCRL2 tool set. The mCRL2 tool set is based in a process algebra. It has the possibility to check the simulation relation.

New features can also be included in the formalism. One issue that we have found is the treatment of recuperations. For instance, let us suppose that we specify the agents:

\[ P_1 = \text{Ob}(a?)[0,10] \triangleright \text{Ob}(b?)[0,10], \quad P_2 = a![0,10] \sqcap b![20,10] \]

and the correct contract \( C = P_1 || P_2 \). Then \( Q = b![20,10] \) is a valid implementation of \( P_2 \). That is, an agent that only implements the recuperation is considered a valid implementation. Intuitively, this is not what we could expect from a service, for instance a vending machine that always refunds the money instead of serving the coffee. To avoid this situation, a line of research is the introduction of probabilities so the normal behaviour has a higher probability against its recuperation. For instance, it should be desirable that the machine delivers the coffee in at least 95% of the interactions instead of refunding the money.

References


