Validation and verification of Web services choreographies by using timed automata

M. Emilia Cambronero*, Gregorio Díaz, Valentín Valero, Enrique Martínez
Departamento de Sistemas Informáticos, Escuela Superior de Ingeniería, Informática de Albacete, Universidad de Castilla-La Mancha, Campus Universitario s/n, 02071 Albacete, Spain

ABSTRACT
We present an approach for the validation and verification of Web services choreographies, and more specifically, for those composite Web services systems with timing restrictions. We use a W3C proposal for the description of composite Web services, WS-CDL (Web Services Choreography Description Language), and we define an operational semantics for a relevant subset of it. We then define a translation of the considered subset of WS-CDL into a network of timed automata, proving that this translation is correct. Finally, we use the UPPAAL tool for the validation and verification of the described system, by using the generated timed automata.

ARTICLE INFO
Article history:
Available online 1 March 2010

Keywords:
Web services
Choreographies
WS-CDL
Timed automata
Formal models

1. Introduction
Web services choreographies provide a way to specify the inter-operation of highly distributed and heterogeneous Web-hosted services. In the last few years various approaches have been taken to describe Web services compositions, such as WS-CDL (Web Services Choreography Description Language) [23], WSCI (Web Service Choreography Interface) [2] or DAML-S [1,24]. With these languages, the interactions and the conditions under which these interactions occur among the different actors in a composite Web service are described, which is the aim of the choreography level, the highest in the Service-Oriented Architecture (SOA). A choreography is therefore a description of the peer-to-peer externally observable interactions that occur between services. The interactions between the participants are therefore described from a global or neutral point of view and not from any specific service perspective. WS-CDL fulfils these requirements, defining a common behavioural contract for all the participants. A WS-CDL specification is an XML document, which can be considered as a contract that all the participants must follow. This contract describes the relevant global definitions, the ordering conditions and the constraints under which messages are exchanged. Each partner can then use this global description to build and test solutions that conform to it. The global specification is in turn enacted by a combination of the resulting local systems, on the basis of appropriate infrastructure support.

By contrast, the orchestration level refers to the business logic that every participant uses, so that it describes the composition at a different level. The execution logic of Web services-based applications is described at the orchestration level by defining their control flows (such as conditional, sequential, parallel and exception handling process) and prescribing the
rules for consistently managing their non-observable data. Thus, the orchestration refers to the automated execution of a workflow, using an execution language such as WS-BPEL [3].

However, the development of composite Web services is still an emerging technique, and in general there is a need for effective and efficient means to abstract, compose, analyze, and evolve Web services within an appropriate time-frame [17]. This paper concentrates on the validation and verification of composite Web services, by using formal techniques. Therefore, in this work we present a technique for the formal verification and validation of Web services choreographies. The choreographies are described in WS-CDL and validated and verified by using formal techniques. We specifically use timed automata as a well-accepted formalism for the description of timed concurrent systems, and thus, we define a translation of a relevant part of WS-CDL into a network of timed automata (NTA). The validation and verification process is then accomplished by using the UPPAAL tool [19], which is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). Thus, one of the main contributions of this work is the formal translation between the WS-CDL choreographies and timed automata. For that purpose, we define a meta-model of the relevant subset of WS-CDL under consideration, and an operational semantics for it. This is a barred operational semantics, in the sense that we use overbarred and underbarred terms to indicate the current state of the system. This barred semantics allows us to see how a syntactic term evolves in a natural way, and also maintain the workunit operator as a single operator, instead of splitting it in some distinct operators.

The translation from the meta-model of WS-CDL to Timed Automata is then formally defined, and afterwards the proof of soundness, to establish that both the operational semantics of a term of the meta-model and the corresponding network of timed automata behave in the same way. This translation is also supported by a tool we have developed, called Web Services Translation tool, WST for short.

Finally, one of the main benefits of the proposed translation is the verification of properties, which can be checked by using the UPPAAL model checker. Among the properties that can be of interest for the designers of composite Web services we can mention the following:

- Invariants: These are conditions that must be true in all the reachable states, for instance deadlock freeness.
- Goals: These are conditions that must be true in all the final states, as, for instance, whether a server sends a confirmation message upon completion of a process of a client request.
- Reachability properties: Which allow us to check whether a given state can be reached or not, for instance, to confirm that the system will always send us an electronic ticket that we have bought by using a Web service.
- Pre and post conditions: These are properties that must be satisfied before (and after, respectively) the beginning (and termination) of certain activities.
- Implication properties: These establish that whenever a property \( p \) holds, eventually another property \( q \) will hold as well.
- For instance, when a server cannot supply a product to the client because it lacks of stock, then the client will not receive the product.

Time restrictions: Another important group of properties is related with time constraints that the system must fulfill. Thus, for instance, we can check that a certain activity is performed upon the expiration of a given time-out.

We have structured the paper as follows: a discussion of related work is shown in Section 2. Section 3 shows a description of the Web Services Choreography Description Language, as well as a barred operational semantics for a relevant subset of it, which is inspired in the textual description of the WS-CDL that can be found in [23]. Timed automata and their semantics are described in Section 4. The translation from WS-CDL to timed automata is defined in Section 5, which is proved to be correct in Section 6. The tool we have developed to support this translation is briefly described in Section 7. A Case Study that illustrates the translation is presented in Section 8, and finally, the conclusions and future work are presented in Section 9.

2. Related work

The developers of WS-CDL claim that its design has been based on a formal language, the \( \pi \)-calculus [22], and that therefore WS-CDL is a particularly well-suited language for describing concurrent processes and dynamic interconnection scenarios. This relationship has been studied in [11], where the authors compare a formalized version of WS-CDL, called global calculus, with the \( \pi \)-calculus. They discuss how the same business protocols can be described by WS-CDL and \( \pi \)-calculus equivalently, as two different ways of describing communication-centered software in the form of formal calculi.

Dong et al. [14] have analyzed orchestrations by means of the Orc language, and they apply the UPPAAL model-checker to verify the described systems. The main difference to our approach is that they work with orchestrations, rather than choreographies. Thus, Orc is a language close to WS-BPEL. Howard Foster et al. [16] also use the orchestration level and WS-BPEL to describe composite Web services. The formalism used by Foster is a Label Transition System (LTS), which is obtained by using Finite State Process (FSP) as an intermediate language. One of the most important contributions of the group led by Foster has been the development of the WS-Engineer framework, an eclipse plugin that implements these techniques. The main difference to our work is that Foster’s work is more generalized, and does not take into account timed behaviours. A more related work is that of Yang Hongli et al. [29], in which WS-CDL is also analyzed by using formal techniques. However, our work covers a wider subset of WS-CDL, which includes the main activity constructions of WS-CDL, variables, error handling, and time-outs in interactions, and we further use a barred operational semantics in order to formalize the
language, maintaining the workunit operator as a single operator, i.e., our meta-model is closer to the syntax of WS-CDL. In [12] the verification of Web services compositions written in WS-CDL is also accomplished by using timed automata, but no formalization is provided either for the WS-CDL semantics or for the translation to timed automata.

There are other related works: Du et al. [15] have defined a formal model, called Nested Web Service Interface Control Flow Automata (NWCFA), which is aimed at the modelling of individual Web services, which form a composition. This formalism focuses on the control flow and service invocation behaviour and uses the technique of assertion-based verification of safety, call stack inspection properties and pre/post conditions of certain service invocations. Sharygina [26] presents a model checking with abstraction techniques for Web services, which translates php implementations into a kripke structure to verify it with SATAbS. There are other works that use Petri nets, in [27] we presented a methodology for the design, verification and validation of composite Web services using WS-CDL as the language for describing Web services interactions, and Petri nets as a formalism that allows us to analyse the described systems. In this previous work we considered timed and prioritized collaborations in composite Web services, so the considered model of Petri nets is a prioritized version of Time Petri nets. Therefore, there are strong differences with this previous work, mainly, we now use Timed Automata as formalism to simulate, validate and verify Web services compositions with timel restrictions, but priorities are not considered in this paper. Timed automata are a widely used formalism for the description of concurrent systems, but specifically for timed concurrent systems, and there are many tools supporting them, as UPPAAL [19], which is a modeling and verification tool based on Timed Automata, which has a high degree of maturity and efficiency. Besides, in this paper we also present a formalization of the syntax and an operational semantics for the considered subset of WS-CDL, so the presented translation into Timed Automata is proved to be correct with respect to this operational semantics.

There are some other works based on Petri nets, Lohman and Kleine [20] use open workflow net models and define a fully-automatic translation of this formalism into abstract BPEL by using the Tools4BPEL framework. The aim of Lohman’s work is, therefore, the automatic generation of BPEL code by using the workflow net formalism as design model, which can be checked of being free of deadlocks and livelocks. This proposal then takes the orchestration viewpoint, whereas we take the choreographic one. Furthermore, we translate WS-CDL specifications into timed automata to verify the requirements and constraints of the system by means of model checking, whereas Lohman’s approach uses as starting point a workflow net model, which is translated into WS-BPEL abstract code. Another important benefit of our approach is that it allows an early validation and verification of the first step in the development of a composite Web service, namely, the choreography design, by using some existing tools, as pi4soa [31] and WST [10].

There are also translations that use algebraic models. Salaün et al. [25] have presented a general proposal for describing and reasoning about choreographies and orchestrations of Web services using this formalism. However, only an informal mapping between WS-BPEL orchestrations and process algebra is presented, and as the authors recognize, a further discussion is required in this work. Nevertheless, the reasoning presented about the use of process algebras in the framework of Web services is commendable. Brogi’s work focusses on the choreography layer, but takes as starting point a different Web services choreography description language (WSCl), which was one of the first proposals for choreography descriptions. However, WS-CDL has a richer expressivity than WSCI, as stated in the comparative work of Cambronero et al. [9]. Yeung [30] has defined a mapping from WS-CDL and BPEL4WS into CSP with the objective of verifying if the obtained orchestrations behave as specified in the corresponding choreography. This mapping is only presented in an informal way, by means of conversion tables, and also the considered subsets of WS-CDL and WS-BPEL are quite basic.

3. WS-CDL

In this subsection we first present a description of the main features of WS-CDL, and an operational semantics for the specific subset of WS-CDL that we use. We will use this semantics in order to establish an equivalence with the NTA that we associate with a WS-CDL model.

3.1. WS-CDL description

A WS-CDL document [23] basically consists of partners, roles, the exchanged information description, choreographies, channels and activities. Partners and roles are used respectively to specify the collaborating entities and the different types of behaviour of these entities, although, for simplicity, we will use partners and roles indistinctly. Choreographies are the main elements of a WS-CDL description. In general, a WS-CDL document contains a hierarchy of choreographies, one of them being the root choreography, while the others are performed by explicit invocation. However, in this paper we will only consider basic WS-CDL specifications, which have only the root choreography.

A choreography has three main parts: the life-line, the exception block, and the finalizer blocks. The life-line contains the activities performed by the choreography in normal conditions. In the event of a failure, the control flow is transferred to the exception block (or the exception is propagated to the enclosing choreography when the exception cannot be handled in its exception block). A finalizer block in a choreography is always associated to an immediately enclosed choreography, and can
only be executed (by explicit invocation) when its corresponding choreography has successfully completed its execution. Obviously, the root choreography will not have any finalizer blocks, so we will omit them in our meta-model.

The collaborative behavior of the participants in a choreography is described by means of activities. These are the actions performed within a choreography, and are divided into three groups: basic activities, ordering structures and workunits. The basic activities are used to establish the variable values (assign), to indicate some inner action of a specific participant (silent_action), or that a participant does not perform any action (noaction), and also to establish an exchange of messages between two participants (interaction). An interaction can be assigned a time-out, i.e., a time to be completed. When this time-out expires (after the interaction was initiated), if the interaction has not completed, the timeout occurs and the interaction finishes abnormally, causing an exception block to be executed in the choreography. Channels are used in the interactions to establish where and how the information is exchanged. They are implicitly considered in our model in the interactions, since they are used to exchange the information from role to role, but we do not introduce a specific syntax for that purpose. An important feature of WS-CDL, namely, channel passing can be regarded as a kind of communication in our meta-model, in the sense that some variables can be used for that purpose, so that channel references can be transferred by means of interactions, and the values of these variables can later be checked in order to launch the corresponding activities for the collaborations that use those channels.

The ordering structures are used to combine activities with other ordering structures in a nested structure to express the ordering conditions under which information within the choreography is exchanged. The ordering structures are the sequence, choice and parallel, with the usual interpretation. Finally, workunits are used to allow the execution of some activities when a certain condition holds. Thus, a workunit encapsulates one activity, which can only be executed if the corresponding guard is evaluated to true. Furthermore, there is another guard in the workunits in order to allow the iteration of the enclosed activity.

### 3.2. Syntax and semantics

We now define the formal syntax and the semantics for the meta-model of the subset of WS-CDL that we use. We call \( \text{Var} \) the set of variable names used in the choreography, the clock variable being one of these variables, which contains the current time, and thus, automatically increases its value as time elapses. We assume that each role type uses its own variable names, the set of variable names used in the choreography, the \( \text{Vars} \).

We now define the formal syntax and the semantics for the meta-model of the subset of WS-CDL that we use. We call \( \text{Var} \) the set of variable names used in the choreography, the clock variable being one of these variables, which contains the current time, and thus, automatically increases its value as time elapses. We assume that each role type uses its own variable names, i.e., a variable name can only be used by a single role type, excepting the clock variable, whose value can be considered as obtained from a time server. For simplicity we only consider non-negative integer variables, although it would not be problematic to extend this assumption to any number of data types. Furthermore, we also consider that each interaction only contains one exchange element, which is used to communicate the value of a variable from one role type to the other.

The specific algebraic language that we use for the activities is defined by the following BNF-notation:

\[
A ::= \text{fail} \mid \text{assign}(r,v,n) \mid \text{noaction}(r) \mid \text{inter}(r_1,r_2,v_1,v_2,t) \mid A:A \mid \\
A \sqcap A \mid A \parallel A \mid \text{workunit}(g,\text{block},g',A)
\]

where \((r,r_1,r_2)\) range over the roletypes of the choreography, \(t \in \mathbb{N} \cup \{\infty\}\), \(v,v_1,v_2\) range over \(\text{Var}\), \(n \in \mathbb{N}\), \(g,g'\) are predicates that use the variable names in \(\text{Var}\), and \(\text{block}\) is a boolean. Given a predicate \(g\), we will call \(\text{Vars}(g)\) the set of variables used in \(g\).

Tables 1 and 2 show the mapping between WS-CDL syntax and the algebraic language. The basic activities are \(\text{fail}, \text{assign}, \text{noaction}\) and \(\text{inter}; \text{fail}\) is used to raise an exception, the control flow is transferred to the exception block, and after that the choreography terminates. The assign operation is used to assign the variable \(v\) at role \(r\) to \(n\), and this is immediate, i.e., it does not take any time to complete; the noaction captures either a silent or internal operation at role \(r\), and this is immediate too. The inter operation is used to capture an interaction between roles \(r_1\) and \(r_2\), with a time-out \(t\) (which can be infinite), where the value of variable \(v_2\) in \(r_2\) is assigned to the value of variable \(v_1\) of \(r_1\). If the time-out expires and the interaction has not been executed, the exception block of the choreography is executed, and after that the choreography terminates. An interaction also fails when the variable \(v_1\) in \(r_1\) is unassigned.

The ordering structures are the sequence, choice and parallel. We also have the workunit operator with the following interpretation: firstly, if some of the variables used in \(g\) are not available, or if \(g\) evaluates to false then, depending on the block attribute the workunit is skipped or it is blocked until \(g\) is evaluated to true. When the guard evaluates to true, the activity inside the workunit is executed, and when it terminates, the repetition condition \(g'\) is evaluated. If some variable used in \(g'\) is not available or if \(g'\) is false, the workunit terminates, otherwise the activity inside it is executed again. The sequence and parallel operators have the usual interpretation.

For the choice, any activity of those enabled in the choice can be executed. We also impose for the block attribute of the workunits that are alternatives of a choice the condition of being true, since in that case we must only consider those workunits whose guard evaluates to true, and it makes no sense to abandon the choice when a workunit guard is false.

A choreography is now defined as a pair \((A_1,A_2)\), where \(A_1\) and \(A_2\) are activities defined by the previous syntax. \(A_1\) is the activity of the life-line of the choreography and \(A_2\) is the activity of its exception block, which can be empty (denoted by \(\emptyset\)), because the exception block is optional.

---

1 **Actually, WS-CDL does not allow the use of shared variables.**

2 **In the sense that it can execute some action at the current instant of time.**
We now introduce the operational semantics for this language, by using both overbarred and underbarred dynamic terms, which are used to capture the current state of the choreography throughout its execution. Before introducing the dynamic terms, we need to consider an extended version of the activity syntax, in which we add the following operator $\text{dinter}(r_1, r_2, v_1, v_2, t, t')$, with $t' \leq t$, called dynamic interaction, which represents an interaction that initially had a time-out $t$ and now has $t'$ time units left before time-out expiration, i.e., if the interaction has not been performed in $t'$ time units a time-out exception is raised, and the control is transferred to the exception block (if present). In the case that there is

<table>
<thead>
<tr>
<th>WS–CDL Term</th>
<th>Meta–model Term</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;noAction roleType=&quot;r&quot;/&gt;</code></td>
<td>noaction(r)</td>
</tr>
<tr>
<td><code>&lt;exceptionBlock ...</code></td>
<td>fail</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td><code>&lt;/exceptionBlock&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;assign roleType=&quot;r&quot;&gt;</code></td>
<td>assign(r,v,n)</td>
</tr>
<tr>
<td><code>&lt;copy name=&quot; &quot;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;source variable=&quot;n&quot;/&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;target variable=&quot;v&quot;/&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/copy&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/assign&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;interaction ...</code></td>
<td>inter(r_1,r_2,v_1,v_2,t)</td>
</tr>
<tr>
<td><code>&lt;participate relationshipType=&quot;...&quot;</code></td>
<td></td>
</tr>
<tr>
<td><code>fromRoleTypeRef=&quot;r1&quot;</code></td>
<td></td>
</tr>
<tr>
<td><code>toRoleTypeRef=&quot;r2&quot;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;exchange ...</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;send variable=&quot;v1&quot;/&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;receive variable=&quot;v2&quot;/&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/exchange&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;timeout time-to-complete=&quot;t&quot;/&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/interaction&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;sequence&gt;</code></td>
<td>$A_1; A_2$</td>
</tr>
<tr>
<td>$A_1$</td>
<td></td>
</tr>
<tr>
<td>$A_2$</td>
<td></td>
</tr>
<tr>
<td><code>&lt;/sequence&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;parallel&gt;</code></td>
<td>$A_1</td>
</tr>
<tr>
<td>$A_1$</td>
<td></td>
</tr>
<tr>
<td>$A_2$</td>
<td></td>
</tr>
<tr>
<td><code>&lt;/parallel&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 2
Mapping of WS-CDL syntax (II).

<table>
<thead>
<tr>
<th>WS-CDL Term</th>
<th>Meta-model Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;workunit name=&quot;nc&quot; guard=&quot;bool expr&quot;? repeat=&quot;bool expr&quot;? block=&quot;True&quot;? &gt; Activity–Notation &lt;/workunit&gt;</td>
<td>workunit(g,True,g',A)</td>
</tr>
<tr>
<td>&lt;workunit name=&quot;nc&quot; guard=&quot;bool expr&quot;? repeat=&quot;bool expr&quot;? block=&quot;False&quot;? &gt; Activity–Notation &lt;/workunit&gt;</td>
<td>workunit(g,False,g',A)</td>
</tr>
<tr>
<td>&lt;choice&gt; Activity–Notation+ &lt;/choice&gt;</td>
<td>$A_1 \square A_2$</td>
</tr>
</tbody>
</table>

no exception block, the choreography terminates abnormally. We will use letters $B, B_1, B_2, \ldots$ to denote activities with the extended syntax, which are used to define the dynamic terms, these are defined by the following BNF-notation:

$$D ::= \overline{B} \mid B \mid D : B \mid B ; D \mid D \square B \mid B \square D \mid \text{workunit}(g, block, g', D)$$

The set of dynamic terms will be called Dterms. The overbars are used to indicate that the corresponding term can initiate its execution, whereas underbarred terms have already finished their execution. Thus, as the activity evolves along its execution the bars are moving throughout the term syntax.

**Example 1.** Consider the activity $A = \text{workunit}(g, true, g', \text{assign}(r, v, 1))$. Its execution starts with the dynamic term $\overline{A}$, from which the guard $g$ is evaluated. If all the variables in $g$ are available, and $g$ becomes true, we reach the dynamic term $D_1 = \text{workunit}(g, true, g', \text{assign}(r, v, 1))$, which means that the assignment of $v$ can now start at role $r$. Otherwise, if some variable needed to evaluate $g$ is not available, or if $g$ is false, as the block condition is true, the activity blocks until $g$ changes its value to true. Once the assignment of $v$ is done, the following dynamic term is reached: $D_2 = \text{workunit}(g, true, g', \text{assign}(r, v, 1))$, from which $g'$ is evaluated. If some variable needed to evaluate $g'$ is not available or $g'$ is false, the workunit ends and the dynamic term $\overline{A}$ is reached. Otherwise, when $g'$ is true, $D_1$ is reached again.

In this example we have used dynamic terms to represent the current state of the system. However, dynamic terms like $B_1 \square B_2$, $\overline{B_1} \square B_2$ and $B_1 \square \overline{B_2}$ correspond to the same state in the system, a state in which any alternative of the choice must be enabled. This means that in some cases the bars can be redistributed on a dynamic term yielding to an equivalent state. Thus, we now define the equivalence relation $\equiv$, as the least equivalence relation satisfying the rules of Table 3. By means of this equivalence relation we can identify those dynamic terms that can be obtained by moving the bars on the terms backwards or forwards, without executing any action and which correspond to the same state in the system. It will also identify the activation of an interaction with the corresponding dynamic interaction that has the whole time-out to complete.

For any dynamic term $D$ we will denote the class of dynamic terms equivalent to $D$ by $[D]_\equiv$, and the set of classes of dynamic terms will be called Cterms.
The rules of Table 3 are immediately intuitive in general. Seq1 is used to activate the first activity of a sequence when the sequence becomes activated. Seq2 allows us to activate $B_2$ when $B_1$ terminates, and Seq3 establishes that once $B_2$ ends, the sequence $B_1; B_2$ ends too. Cho1 and Cho2 allow us to activate either alternative of a choice, while Cho3 and Cho4 establish that once the selected alternative terminates the choice itself ends, too. Par1 is used to activate both arguments in a parallel activity, and Par2 establishes that, when both argument activities terminate, the parallel activity terminates, too. \(\text{Inter}\) identifies the activation of an interaction with the dynamic interaction having its whole time-out to be executed. The last three rules establish that $\equiv$ is actually a congruence.

**Definition 1 (Initial and final dynamic terms).** Given a dynamic term $D$, we say that $D$ is initial (resp. final), denoted by $\text{init}(D)$ (resp. $\text{final}(D)$), when there exists an extended activity $B$ such that $B \in [D]_\text{init}$ (resp. $B \in [D]_\text{final}$). In such a case we will say that the class $[D]_\text{init}$ is initial (resp. final) too.

For instance, the terms $\text{assign}(r, v, n)$, $\text{assign}(r, v, n) \circ \text{noaction}(r)$ and $\text{assign}(r, v, n) \parallel \text{noaction}(r)$ are all initial.

A choreography is executed within the context of the variables defined in it, where a context $\mu$ is defined as a function $\mu : \text{Var} \to \mathbb{N} \cup \{\epsilon\}$, which assigns a value to every variable, where unavailable variables are assigned the $\epsilon$ value. We denote the set of possible contexts of a choreography by $\text{Contexts}$. The initial context, denoted by $\mu_0$, is that defined by assigning $\epsilon$ to all the variables in the choreography, except the clock, which is assigned to $0$. Given a context $\mu$, a variable $v$ and an integer arithmetic expression $n$, we denote by $\mu[v/n]$ the context obtained from $\mu$ by changing the value of $v$ to $n$. Given a predicate $g$ and a context $\mu$, we will write $\text{sat}(\mu, g)$ when $\forall v \in \text{Vars}(g), \mu(v) \neq \epsilon$, and $g$ evaluates to true under $\mu$.

Time elapsing is captured by means of the following function, which ages a class of dynamic terms by one time unit:

**Definition 2 (Aging function).** The function $\text{aging} : \text{CDterms} \to \text{CDterms}$ is defined in a structural way, as follows:

For any dynamic terms $D, D_1, D_2$:

1. If $\text{final}(D)$, then $\text{aging}([D]_\text{final}) = [D]_\text{final}$.
2. $\text{aging}([\text{fail}]_\text{final}) = [\text{fail}]_\text{final}$.
3. $\text{aging}([\text{assign}(r, v, n)]_\text{final}) = [\text{assign}(r, v, n)]_\text{final}$.
4. $\text{aging}([\text{noaction}]_\text{final}) = [\text{noaction}]_\text{final}$.
5. For $t' > 0$:
   - $\text{aging}([\text{dinter}(r_1, r_2, v_1, v_2, t, t')]_\text{final}) = [\text{dinter}(r_1, r_2, v_1, v_2, t, t' - 1)]_\text{final}$, where we take $\infty - 1 = \infty$.
6. $\text{aging}([\text{dinter}(r_1, r_2, v_1, v_2, 0)]_\text{final}) = [\text{fail}]_\text{final}$.
7. $\text{aging}([\text{workunit}(g, \text{block}, g', B)]_\text{final}) = [\text{workunit}(g, \text{block}, g', B')]_\text{final}$, with $B'$ such that $B' \in \text{aging}([B]_\text{final})$.
8. $\text{aging}([\text{workunit}(g, \text{block}, g', D)]_\text{final}) = [\text{workunit}(g, \text{block}, g', D')]_\text{final}$, with $D' \in \text{aging}([D]_\text{final})$.
9. If $\neg\text{final}(D)$ : $\text{aging}([D; B]_\text{final}) = [D'; B]_\text{final}$ and $\text{aging}([B; D]_\text{final}) = [B; D']_\text{final}$, with $D' \in \text{aging}([D]_\text{final})$.
10. If $\neg\text{init}(D)$ and $\neg\text{final}(D)$ : $\text{aging}([B \circ D]_\text{final}) = [B \circ D']_\text{final}$, and $\text{aging}([D \circ B]_\text{final}) = [D' \circ B]_\text{final}$, with $D' \in \text{aging}([D]_\text{final})$. 

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Equivalence rules for dynamic terms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Seq1)</td>
<td>$B_1 ; B_2 \equiv B_1 ; B_2$</td>
</tr>
<tr>
<td>(Seq2)</td>
<td>$B_1 ; B_2 \equiv B_1 ; B_2$</td>
</tr>
<tr>
<td>(Seq3)</td>
<td>$B_1 ; B_2 \equiv B_1 ; B_2$</td>
</tr>
<tr>
<td>(Cho1)</td>
<td>$B_1 \parallel B_2 \equiv B_1 \parallel B_2$</td>
</tr>
<tr>
<td>(Cho2)</td>
<td>$B_1 \parallel B_2 \equiv B_1 \parallel B_2$</td>
</tr>
<tr>
<td>(Cho3)</td>
<td>$B_1 \parallel B_2 \equiv B_1 \parallel B_2$</td>
</tr>
<tr>
<td>(Cho4)</td>
<td>$B_1 \parallel B_2 \equiv B_1 \parallel B_2$</td>
</tr>
<tr>
<td>(Par1)</td>
<td>$B_1 ; B_2 \equiv B_1 ; B_2$</td>
</tr>
<tr>
<td>(Par2)</td>
<td>$B_1 ; B_2 \equiv B_1 ; B_2$</td>
</tr>
</tbody>
</table>

\[ \text{aging}(r, v, n) = \text{aging}(r, v, n) \parallel \text{noaction}(r) \text{ and } \text{aging}(r, v, n) \parallel \text{noaction}(r) \text{ are all initial.} \]
(11) If init(D) ∧ ¬final(D) : aging([B □ D]₁⁺) = [B′ □ D′]₁⁺, and
aging([D □ B]₁⁺) = [D′ □ B′]₁⁺, with D′ ∈ aging([D]₁⁺) and B′ such that B′ ∈ aging([B]₁⁺).
(12) If ¬final(D₁) ∧ ¬final(D₂) : aging([D₁ || D₂]₁⁺) = [D₁’ || D₂’]₁⁺,
with D₁’ ∈ aging([D₁]₁⁺) and D₂’ ∈ aging([D₂]₁⁺).
(13) If final(D₁) ∧ ¬final(D₂) : aging([D₁ || D₂]₁⁺) = [D₁’ || D₂’]₁⁺, and
aging([D₂ || D₁]₁⁺) = [D₂’ || D₁’]₁⁺, with D₂’ ∈ aging([D₂]₁⁺).

From this definition, we can see that when an interaction expires (point 6) we obtain a failure, which will allow us to execute the exception block (except if we find ourselves facing a choice with some other possible alternatives, as we will see later). The passage of time for dynamic interactions is captured by means of point 5. We can also see that the passage of time over an activated workunit is passed to the activity inside it (point 7), since we consider that the first activity of the workunit is in some sense activated once the workunit has been reached (although it can only be executed when the guard condition is true). Point 11 also requires some explanations, in this case the passage of time over an activated choice is passed to both argument activities. As the remaining points are quite self-explanatory, we shall omit further comment.

Therefore, with the function aging we transform one class into another, capturing the elapse of one time unit. However, in some cases we do not allow the passage of time, since some movement must be made immediately. This occurs, for instance, when an exception has been raised; in this case the exception block is immediately executed. Furthermore, in general, not only time elapsing, but all the possible evolutions of a class depend on the current context. Hence, we introduce the so-called contextual activity terms, as pairs (D₁⁺, μ), where D is a dynamic term and μ a context.

We now define a boolean function elapse, which indicates to us whether time can or cannot elapse for any contextual activity term.

**Definition 3 (Function elapse).** The function elapse : CDterms × Contexts → Boolean is defined in a structural way, as follows:

For any dynamic terms D₁⁺, D₂⁺ and any context μ :
(1) If final(D), then elapse([D]₁⁺, μ) = true.
(2) elapse([fail]₁⁺, μ) = false.
(3) elapse([assign(r, v, n)]₁⁺, μ) = true.
(4) elapse([noaction(f)]₁⁺, μ) = true.
(5) If μ(v₁) ≠ ϵ : elapse([dinter(r₁, r₂, v₁, v₂, t, t’)]₁⁺, μ) = true.
(6) If μ(v₁) = ϵ : elapse([dinter(r₁, r₂, v₁, v₂, t, t’)]₁⁺, μ) = false.
(7) elapse([workunit(g, block, g’, B)]₁⁺, μ) = block.
(8) If ¬final(D) : elapse([workunit(g, block, g’, D)]₁⁺, μ) = elapse([D]₁⁺).
(9) If final(D) : elapse([workunit(g, block, g’, D)]₁⁺, μ) = false.
(10) If ¬final(D) : elapse([B; D]₁⁺, μ) = elapse([B; D]₁⁺, μ) = elapse([D]₁⁺, μ).
(11) If ¬init(D) ∧ ¬final(D) : elapse([B □ D]₁⁺, μ) = elapse([D □ B]₁⁺, μ) = elapse([D]₁⁺, μ).
(12) If init(D) ∧ ¬final(D) : elapse([B □ D]₁⁺, μ) = elapse([D □ B]₁⁺, μ) = elapse([B]₁⁺, μ) ∨ elapse([D]₁⁺, μ).
(13) If ¬final(D₁) ∧ ¬final(D₂) : elapse([D₁ || D₂]₁⁺, μ) = elapse([D₁]₁⁺, μ) ∧ elapse([D₂]₁⁺, μ).
(14) If final(D₁) ∧ ¬final(D₂) : elapse([D₁ || D₂]₁⁺, μ) = elapse([D₂ || D₁]₁⁺, μ) = elapse([D₂]₁⁺, μ).

To check that elapse is a well defined function is immediate. By means of elapse the passage of time is not allowed when an exception has been raised (point 2), except in the case of the failure being caused by an alternative of a choice, since some other alternatives could be allowed. Thus, for instance, if an interaction with a time-out has expired, this interaction cannot be executed, but there may be some other possible alternatives in the choice that are still enabled. In point 6, we can also see that, when the source variable of an interaction is unassigned, time cannot elapse, because we immediately raise an exception. In the case of an activated workunit (point 7), depending on the block attribute we can wait or not, and when the activity of the workunit terminates, the repetition condition g’ must be evaluated immediately, so no time can elapse here (point 9). For an activated choice (point 12) we allow the passage of time when at least one alternative does allow it. Thus, in a choice we may have some interactions with time-outs that have expired, but the choice may still offer some alternatives. However, in the case of a parallel, time cannot elapse when one alternative does not allow this.

**Definition 4.** We define a dynamic choreography term as a pair of one of the following forms: ([D]₁⁺, A₂) or (A₁, [D]₁⁺), where [D]₁⁺ corresponds to the activity in execution in the choreography (the life-line or its exception block), and A₂ can be empty.

We also define a contextual dynamic choreography term, as a pair ([C], μ), where C is a dynamic choreography term and μ is a context.
Table 4
Transition rules for contextual activity terms (I).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Clock)</td>
<td>(\text{elapse}(\Delta t, \mu))</td>
</tr>
<tr>
<td></td>
<td>((\Delta t, \mu) \rightarrow 1 (\Delta t', \mu))</td>
</tr>
<tr>
<td>(Fail)</td>
<td>(\text{fail}(\Delta t, \mu))</td>
</tr>
<tr>
<td></td>
<td>((\text{fail}(\Delta t, \mu) \rightarrow (\text{fail}(\Delta t, \mu)))</td>
</tr>
<tr>
<td>(Assign)</td>
<td>(\text{assign}(r, v, n))</td>
</tr>
<tr>
<td></td>
<td>((\text{assign}(r, v, n), \mu) \rightarrow (\text{assign}(r, v, n), \mu(\Delta t, \mu)))</td>
</tr>
<tr>
<td>(Noact)</td>
<td>(\text{noaction}(r))</td>
</tr>
<tr>
<td></td>
<td>((\text{noaction}(r), \mu) \rightarrow (\text{noaction}(r), \mu))</td>
</tr>
<tr>
<td>(Int1)</td>
<td>(\text{inter}(r, t_1, t_2, v_1, v_2, f))</td>
</tr>
<tr>
<td></td>
<td>((\text{inter}(r, t_1, t_2, v_1, v_2, f), \mu) \rightarrow (\text{inter}(r, t_1, t_2, v_1, v_2, f), \mu(\Delta t, \mu)))</td>
</tr>
<tr>
<td>(Int2)</td>
<td>(\text{inter}(r, t_1, t_2, v_1, v_2, f))</td>
</tr>
<tr>
<td></td>
<td>((\text{inter}(r, t_1, t_2, v_1, v_2, f), \mu) \rightarrow (\text{inter}(r, t_1, t_2, v_1, v_2, f), \mu(\Delta t, \mu)))</td>
</tr>
<tr>
<td>(Work1)</td>
<td>(\text{workunit}(g, \text{block}, g', \text{B}))</td>
</tr>
<tr>
<td></td>
<td>((\text{workunit}(g, \text{block}, g', \text{B}), \mu) \rightarrow (\text{workunit}(g, \text{block}, g', \text{B}), \mu))</td>
</tr>
<tr>
<td>(Work2)</td>
<td>(\text{workunit}(g, \text{block}, g', \text{B}))</td>
</tr>
<tr>
<td></td>
<td>((\text{workunit}(g, \text{block}, g', \text{B}), \mu) \rightarrow (\text{workunit}(g, \text{block}, g', \text{B}), \mu(\Delta t, \mu)))</td>
</tr>
<tr>
<td>(Work3)</td>
<td>(\neg\text{sat}(\mu, g))</td>
</tr>
<tr>
<td></td>
<td>((\neg\text{sat}(\mu, g), \mu) \rightarrow (\neg\text{sat}(\mu, g), \mu))</td>
</tr>
<tr>
<td>(Work4)</td>
<td>(\text{workunit}(g, \text{false}, g', \text{B}))</td>
</tr>
<tr>
<td></td>
<td>((\text{workunit}(g, \text{false}, g', \text{B}), \mu) \rightarrow (\text{workunit}(g, \text{false}, g', \text{B}), \mu))</td>
</tr>
<tr>
<td>(Work5)</td>
<td>(\text{workunit}(g, \text{block}, g', \text{D}))</td>
</tr>
<tr>
<td></td>
<td>((\text{workunit}(g, \text{block}, g', \text{D}), \mu) \rightarrow (\text{workunit}(g, \text{block}, g', \text{D}), \mu))</td>
</tr>
<tr>
<td>(Work6)</td>
<td>(\text{workunit}(g, \text{block}, g', \text{D}))</td>
</tr>
<tr>
<td></td>
<td>((\text{workunit}(g, \text{block}, g', \text{D}), \mu) \rightarrow (\text{workunit}(g, \text{block}, g', \text{D}), \mu(\Delta t, \mu)))</td>
</tr>
<tr>
<td>(Work7)</td>
<td>(\neg\text{sat}(\mu, g))</td>
</tr>
<tr>
<td></td>
<td>((\neg\text{sat}(\mu, g), D) \rightarrow (\neg\text{sat}(\mu, g), D))</td>
</tr>
</tbody>
</table>

Given a choreography \(C = (A_1, A_2)\), the initial contextual dynamic term of \(C\) is\(^3\) \((\overrightarrow{A_1}, A_2, \mu_0)\).

In Tables 4 and 5, we introduce the rules that define the transitions for the contextual activity terms, where we can see that we have two types of transition:

- \((\Delta t, \mu) \rightarrow 1 (\Delta t', \mu')\) : which represents the passage of one time unit.
- \((\Delta t, \mu) \rightarrow a (\Delta t', \mu')\) : which represents the execution of some basic activity \(a\) or an empty movement (denoted by \(a = \emptyset\)). In this case no time elapses.

In rules Par1 and Par2 of Table 5 we use the notation \((\Delta t, \mu) \rightarrow \text{fail} (\Delta t', \mu')\) to mean that no transition labelled with \text{fail} can be executed from \((\Delta t, \mu)\). Rule \text{Clock} is used to capture the passage of one time unit. Rules \text{Fail}, \text{Assign} and \text{Noact} are evident, whereas \text{Int1} captures the execution of an activated interaction, when the source variable has a value assigned. Otherwise, rule \text{Int2} is used to raise an exception. Rules \text{Work1} to \text{Work7} establish the semantics of workunits, according to the interpretation described previously. Rules \text{Seq1} to \text{Seq4} capture the semantics of the sequence operator, while \text{Choi1} to \text{Choi7} define the semantics of the choice. Rules Choi1-2 are used to resolve the choice when one argument activity can

\(^3\) We will write contextual dynamic choreography terms as triples, by omitting the parentheses for the dynamic choreography term.
execute a non-fail movement. Once the choice has been decided by executing a movement of one of its argument activities, this activity continues executing until completion (Choi4–5). Observe that the failure of the activity in execution is propagated to the choice, which fails as well (rules Choi6–7). Rule Choi3 states that a fail movement can only be executed in a choice when both arguments are able to do so. Accordingly, when an alternative fails (for instance, a time-out of an interaction has expired), this alternative is not considered for execution, but the other ones can proceed (in fact, we allow time elapsing in that case, because we may have some other interactions that can be executed some time later). Finally, rules Par1–2 capture the (independent) parallel execution of the argument activities of a parallel operator, and Par3–4 are used to raise an exception when one component is able to do so.

The rules for choreographies are those introduced in Table 6, which capture the evolution of contextual dynamic choreography terms, as an extension of the contextual activity terms. We then define the labelled transition system of a choreography $C = (A_1, A_2)$ as that obtained by the application of these rules starting from $q_0 = ([A_1]_\infty | A_2, \mu_0)$, and we call timed traces the concatenation of both actions and delays that can be executed from the initial state. Timed traces are denoted by letters $a, a' \in (\mathbb{N} \cup \text{Act})^*$, where the concatenation of $n$ consecutive delay transitions in a row is considered as a single element $n$ in the trace, $n \in \mathbb{N}$, and Act is the set of action names, including $\emptyset$.

We can also introduce the so-called contextual timed traces, namely, the timed traces obtained for a dynamic activity term, but considering any possible context change throughout its evolution, except in the clock variable, which is assumed to increase its value indefinitely.

**Definition 5 (Contextual timed traces).** Let $D$ be a dynamic activity term. We define the set of contextual timed traces of $|D|_\infty$ as follows:
Transition rules for choreographies.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cor1</td>
<td>$(</td>
</tr>
<tr>
<td>Cor2</td>
<td>$(A_1</td>
</tr>
<tr>
<td>Cor3</td>
<td>$(</td>
</tr>
<tr>
<td>Cor4</td>
<td>$(</td>
</tr>
<tr>
<td>Cor5</td>
<td>$(</td>
</tr>
<tr>
<td>Cor6</td>
<td>$(</td>
</tr>
</tbody>
</table>

\[\text{Cor5} \quad \text{Cor6}\]

\begin{align*}
\text{Ctrl}(|D|_\mu) &= \{\varepsilon\} \cup \{|s| \in (\mathbb{N} \cup \text{Act})^+ | (|D|_\mu, \mu) \xrightarrow{S_1} (|D'|_\mu, \mu_1), \text{ and for all } i \in \{1, \ldots, \text{length}(s)\}, (|D|_\mu, \mu_i) \xrightarrow{S_{i+1}} (|D'|_\mu, \mu_{i+1}), \text{ for any contexts } \mu, \\
&\quad\text{clocks, a finite set of non-negative integer-valued variables } a, b, \ldots \text{ standing for actions. We will use letters } r, r', \ldots \text{ to denote sets of clocks. We will denote by Assigns the set of possible assignments, Assigns = \{V := \text{expr} | V \in \mathcal{V}\}, \text{ where expr are arithmetic expressions using naturals and variables. Letters } s, s', \ldots \text{ will be used to represent a set of assignments.}\
&\quad\text{A guard or invariant condition is a conjunctive formula of atomic constraints of the form: } x \sim n, x - y \sim n, v \sim n \text{ or } v - w \sim n, \text{ for } x, y \in \mathcal{C}, v, w \in \mathcal{V}, ~\sim \in \{\leq, <, =, >\} \text{ and } n \in \mathbb{N}. \text{ The set of guard or invariant conditions will be denoted by } \mathcal{G}, \text{ ranged over by } g, g', \ldots \}.
\end{align*}

4. Timed automata

A timed safety automaton, or simply timed automaton (TA) \cite{4,5} is essentially a finite automaton extended with real-valued variables. These variables model the logical clocks in the system, and are initialized to zero when the system is started. They then increase their value synchronously as time elapses, at the same rate. In the model there are also clock constraints, which are guards on the edges that are used to restrict the behaviour of the automaton, since a transition represented by an edge can only be executed when the clock values allow the guard condition to be satisfied. Nevertheless, transitions are not forced to execute when their guards are true, the automaton being able to stay at a location without executing any transition, unless an invariant condition is associated with that location. In this case, the automaton may remain at that same location as long as the invariant condition is satisfied. Additionally, the execution of a transition can be used to reset some clocks of the automaton.

In the timed automata model that we consider we have also non-negative integer variables and urgent edges. The variables can be assigned a value when executing an edge, and their values can be checked in the guards and invariants. Urgent edges inhibit time elapsing when they are enabled, and in the case of conflict urgent edges are fired first.

**Definition 6 (Timed automaton).** We consider a finite set of real-valued variables \(C\) ranged over by \(x, y, \ldots\) for clocks, a finite set of non-negative integer-valued variables \(V\), ranged over by \(v, w, \ldots\) and a finite alphabet \(\Sigma\) ranged over by \(a, b, \ldots\) for actions. We will use letters \(r, r', \ldots\) to denote sets of clocks. We will denote by Assigns the set of possible assignments, \(\text{Assigns} = \{V := \text{expr} | V \in \mathcal{V}\}\), where \(\text{expr}\) are arithmetic expressions using naturals and variables. Letters \(s, s', \ldots\) will be used to represent a set of assignments.

A guard or invariant condition is a conjunctive formula of atomic constraints of the form: \(x \sim n, x - y \sim n, v \sim n \text{ or } v - w \sim n\), for \(x, y \in C, v, w \in V, ~\sim \in \{\leq, <, =, >\} \) and \(n \in \mathbb{N}\). The set of guard or invariant conditions will be denoted by \(\mathcal{G}\), ranged over by \(g, g', \ldots\).

A timed automaton is a tuple \((N, n_0, E, I)\), where \(N\) is a finite set of locations (nodes), \(n_0 \in N\) is the initial location, \(E \subseteq N \times \mathcal{G} \times \Sigma \times \mathcal{P}(\text{Assigns}) \times 2^C \times N\) is the set of edges, where the subset of urgent edges is called \(E_u \subseteq E\), and they will graphically be distinguished as they will have their arrowhead painted in white. \(I : N \rightarrow \mathcal{G}\) is a function that assigns invariant conditions (which could be empty) to locations.

We will write \(n \xrightarrow{g,a,r} n'\) to denote \((n, g, a, s, r, n') \in E\), and \(n \xrightarrow{g,a,r} u n'\) when \((n, g, a, s, r, n') \in E_u\).
Definition 7 (Timed automaton semantics). Let \( A = (N, n_0, E, I) \) be a timed automaton. The semantics of \( A \) is defined as the timed labelled transition system \((Q, q_0, \rightarrow)\), where:

- \( Q \subseteq N \times (\mathbb{R}_{\geq 0}^{+} \times \mathbb{N}^+) \) (set of states),
- \( q_0 = (n_0, \emptyset) \in Q \) is the initial state, where \( \emptyset \) is the valuation that assigns every clock to zero and every integer variable to \( \epsilon \) (a special natural value representing uninitialized variables),
- \( \rightarrow \subseteq (Q \times \mathbb{R}_{\geq 0}^{+} \times Q) \cup (Q \times \mathbb{R}_{\geq 0}^{+} \times Q) \) (delay and action transitions).

Action transitions are in the form \((q, a, s, q')\), for \( a \in \Sigma \) and \( s, q' \in \mathcal{P}(\text{Assigns}) \), denoted by \( q \xrightarrow{a, s} q' \), and are defined by the following rule:

\[
(n, u) \xrightarrow{a, s} (n', u') \text{ if and only if one of the following conditions holds:}
\]

- There is an edge \( n \xrightarrow{a, s} n' \), such that \( u \in g, u' = (u(s))_l, u' \in I(n') \), and there is no enabled urgent edge leaving \( n \), i.e., there is no edge \( n \xrightarrow{a, s} n'' \), such that \( u \in g' \), and \((u(s'))_l \in I(n'')\).
- There is an urgent edge \( n \xrightarrow{a, s} n' \), such that \( u \in g, u' = (u(s))_l, \text{ and } u' \in I(n') \).

Delay transitions are in the form \((n, d, q')\), for \( d \in \mathbb{R}_{\geq 0}^{+} \), denoted by \( q \xrightarrow{d} q' \), and are defined by the following rule:

\[
(n, u) \xrightarrow{d} (n, u + d) \text{ if and only if the following conditions hold:}
\]

- \((u + d) \in I(n)\), for all \( d' \leq d, d' \in \mathbb{R}_{\geq 0}^{+}\).
- There is no urgent edge leaving \( n \) that can be enabled in this period of time: there is no edge \( n \xrightarrow{a, s} n'' \), such that for some \( d' < d, d' \in \mathbb{R}_{\geq 0}^{+} \), \((u + d') \in g' \), and \((u + d')(s))_r \in I(n'').\]

A concurrent system is usually modelled by a set of timed automata running in parallel. A Network of Timed Automata (NTA) is then defined as a set of timed automata that run simultaneously, using the same set of clocks and variables, and synchronizing on the common actions. In the following definition we distinguish two types of action: internal and synchronization actions. Internal actions can be executed by the corresponding automata independently, and they will be ranged over the letters \( a, b, \ldots \) We will assume, for simplification, that urgent edges can only be internal. Synchronization actions, however, must be executed simultaneously by two automata, and they will always be non-urgent. Synchronization actions are ranged over letters \( m, m', \ldots \) and come from the synchronization of two actions \( m \) and \( m' \) executed from two different automata. The operational semantics of a network of timed automata is then defined in a straightforward way, as a natural extension of Definition 7.

Definition 8 (Semantics of an NTA). Let \( A_i = (N_i, n_{i0}, E_i, I_i) \), \( i = 1, \ldots, k \) be a set of timed automata. A state of the NTA \( \{A_1, \ldots, A_k\} \) is a pair \((\pi, u)\), where \( \pi = (n_1, \ldots, n_k) \), with \( n_i \in N_i \), and \( u \) is a valuation for the clocks and variables in the system.

There are three rules defining the semantics of a NTA:

- \((\pi, u) \xrightarrow{d} (\pi, u + d)\) (delay rule) if and only if \( u + d' \in I_i(n_i) \), for all \( i = 1, \ldots, k \) and for all \( d' \leq d, d' \in \mathbb{R}_{\geq 0}^{+}\).
- \((\pi, u) \xrightarrow{a} (\pi', u')\) (internal action rule) if and only if one of the following conditions holds:
  - There is an edge \( n_i \xrightarrow{a, s} n_i' \), for some \( i \in \{1, \ldots, k\} \), such that \( n_j' = n_j \), for all \( j \neq i, u \in g, u' = (u(s))_l, u' \in \bigwedge_{h=1,\ldots,k} h_i(n_h) \), and there is no enabled urgent edge for any other node, i.e., for all \( j \neq i \) there is no edge \( n_j \xrightarrow{a, s} n_j'' \), such that \( u \in g' \), and \((u(s'))_l \in I(n_j'')\).
  - There is an urgent edge \( n_i \xrightarrow{a, s} n_i' \), for some \( i \in \{1, \ldots, k\} \), such that \( n_j' = n_j \), for all \( j \neq i, u \in g, u' = (u(s))_l, u' \in \bigwedge_{h=1,\ldots,k} h_i(n_h) \).
- \((\pi, u) \xrightarrow{m} (\pi', u')\) (synchronization rule) if and only if there exist \( i, j \), \( i \neq j \), such that:
  1. \( n_j = n_0 \), for all \( h \neq i, h \neq j \).
  2. There exists two edges \( n_i \xrightarrow{a, s} n_i' \) and \( n_j \xrightarrow{a, s} n_j' \), such that \( u \in g_i \land g_j, u' = ((u(s_i))_l)_l, u' \in \bigwedge_{h=1,\ldots,k} h_i(n_h) \).
  3. \( u' \in \bigwedge_{h=1,\ldots,k} h_j(n_h) \).
  4. There is no enabled urgent edge from any other node: for all \( k \) such that \( k \neq i, k \neq j \), there is no edge \( n_k \xrightarrow{a, s} n_k' \), such that \( u \in g' \), and \((u(s'))_l \in I(n''_k)\).

\footnote{We will not use urgent edges in synchronizations.}
From this definition, we can easily define the timed traces of an NTA as the sequences of both delays and actions \( t \in (\mathbb{R}_+^* \cup \Sigma)^* \) that can be obtained from its initial state. We now define a V-context as any variation of a given context in which the variables in the set \( V \subseteq \mathcal{V} \) may have changed their values, but all the remaining variables and all the clocks keep their values. We then define the V-contextual timed traces of an NTA as the sequences of both delays and actions that can be obtained for this NTA considering any possible intermediate V-context, i.e., for any reachable state of the NTA we do not only consider those timed traces that are reachable from it, but also those that could be obtained by changing some specific variable values in the intermediate contexts in the sequence, even in the initial one. In the case of a network of timed automata having one or more final locations (in the sense that we are only interested in the traces obtained until reaching one of these locations) we will call terminal traces to those V-contextual timed traces that terminate in one of those final locations, and after that, only time elapsing is possible.

5. Translating WS-CDL documents into timed automata

A function \( \varphi : \text{Activities} \times \mathcal{P}_x (\mathcal{C}) \times \mathcal{N} \rightarrow \text{NTA} \times \mathcal{P}_x (\mathcal{C}) \) is first defined which associates an NTA to every activity. The main argument of this function is the activity for which the translation is made, but it has two additional arguments: one set of clocks and one location. The set of clocks indicates the clocks that must be reset just before finishing the execution of the generated timed automata (for compositional purposes). The location is used to transfer the control flow there in the event of a failure.

We will denote by \( \varphi_1 (A, C, l) \) the first projection of \( \varphi \), i.e. the obtained NTA, and by \( \varphi_2 (A, C, l) \) its second projection, i.e. the set of clocks that should be reset when using this NTA compositionally.

Thus, given a choreography \( C = (A_1, A_2) \), we define its associated NTA as follows (Fig. 3):

- We first create a location ‘de’, which we call the “double exception location”, which is used as the location to which the control flow is transferred in the event of a failure within the exception activity \( A_2 \). We then generate \( \varphi_2 (A_2, \emptyset, \text{de}) \).
- We now create the exception location ‘e’, where the control flow is transfer in the event of failure in \( A_1 \), and then, we generate \( \varphi (A_1, \emptyset, e) \).
- We connect the exception location ‘e’ with the initial location of \( \varphi_1 (A_2, \emptyset, \text{de}) \) by means of an urgent edge, which must reset all the clocks in \( \varphi_2 (A_2, \emptyset, \text{de}) \).

Figs. 1 and 2 show how the function \( \varphi \) is defined for the different activities, where we can observe that all the obtained automata have both one initial and one final location, this property being preserved by all the constructions. Furthermore, we can see that according to the previous description, in the event of a failure, all of these constructions transfer the control flow to the location indicated as third parameter in the function \( \varphi \). The clocks indicated as third parameter in all the edges reaching the final location are also reset in this situation.

We omit a formal definition of the NTA produced as result of the application of \( \varphi \), as they can easily be deduced from both figures.

Let us now describe briefly how the translation works for the different activities:

- **noaction, fail and assign**: these have a simple translation, as we only have to introduce an edge connecting the initial location with the final one (the exception location in the case of fail). Notice that in the case of fail, the edge is urgent, since no time can elapse when a fail action can be executed. In the assign action we can observe that we need to introduce the corresponding assignment operation in the timed automaton.

- **inter \((r_1, r_2, v_1, v_2, l)\)**: in this case three edges must be considered, one for the interaction execution, which must be performed within the indicated time interval, and only when \( v_1 \) has a value assigned, and two additional edges to capture the two possible cases of failure: time-out expiration (captured by using a location invariant) and \( v_1 \) unassigned (this edge must be urgent). Notice that when \( t = \infty \) the time-out edge would not be introduced.

- **A1: A2**: we first obtain the corresponding NTA both for \( A_1 \) and \( A_2 \), as indicated in Fig. 1, and then, we only need to collapse in a single location the final node of \( \varphi_1 (A_1, \emptyset, l) \) and the initial node of \( \varphi_1 (A_2, \mathcal{C}, l) \). Notice that all the edges reaching this node must reset all the clocks in \( \varphi_2 (A_2, \mathcal{C}, l) \), and also that the set of clocks to be reset when using the generated NTA is that of \( A_1 \).

- **A1 ∥ A2**: we first obtain the corresponding NTA both for \( A_1 \) and \( A_2 \), as indicated in Fig. 1, and then, we add three new locations and the edges indicated in the figure, which are used to enforce the simultaneous initialization and termination of both activities, by means of a new synchronization channel \( c \). We also add a new variable \( v_e \), initialized to 0, which is used to prevent the execution of further transitions of one of the automata when the other one has failed. Thus, we add the guard condition \( (v_e = 0) \) to every edge of both automata, and also the invariants \( l \) are replaced by \( (l \lor v_e = 1) \), to avoid the time lock of the system when a fail has been executed. Furthermore, the assignment \( v_e = 1 \) is now included in every fail edge of both automata.

- **workunit \((g, \text{block}, g^*, A)\)**: we have distinguished two cases, depending on the block value, the difference being that when block is false, there is an urgent edge connecting the new initial location with the new final location, labelled with the action \( t \), which resets the clocks in \( \mathcal{C} \). Notice that in both cases if \( g \) is evaluated to true, the control flow is immediately transferred to the initial location of \( \varphi_1 (A, \emptyset, l) \) by means of another urgent edge, and also that upon

---

5 Including the empty trace \( \epsilon \).
termination of $A$ the repetition guard $g'$ is immediately checked in order to decide whether $A$ should be repeated or the control should be transferred to the new final location.

- **Choice**: the choice operator has a semantics that allows any alternative to proceed by executing any of its enabled actions, which generates some problems in the translation, especially in the case of workunits as alternatives of a choice. We have previously imposed the restriction for these workunits that are alternatives of a choice to have their block argument equals to true, but we need to impose some additional restrictions on them in order to define this translation. The first additional restriction that we consider is that their first activity must be an interaction (it would not be problematic to assume that this activity is either a `noaction` or an `assign`, but in such a case the translation would be slightly different from that shown in the figure).

We also impose that no parallel activity appears as alternative in a choice, to avoid a rather large distinction of subcases. Then, to simplify the description this case is not considered.

![Diagram](image)

**Fig. 1.** From WS-CDL to NTA (I).
With these assumptions we define the translation for the choice operator by unfolding all the inner choices it can contain, i.e., we define the translation for a general choice in which we may have as alternatives the following activities: assign, noaction, inter and workunit, possibly in a sequence with any other operator.

Fig. 2 shows how this translation is made for a general choice in which we have all of these activities as alternatives. However, notice that a choice can also fail, but only in the case that all the alternatives fail. This means, for instance, that if we have either an assign or a noaction as one alternative of the choice, no fail action is possible. Then, in the case
of a choice with no assign and no noaction as alternatives, we must consider the two possible cases of failure: either the maximum time-out \( M \) of all the alternative interactions has expired, with \( M = \text{Max}(t_1, \ldots, t_3, t'_1, \ldots, t'_{4d}) \), or no source variable of these interactions has a value assigned. In Fig. 4 we depict the two urgent edges that we should add in this case.\(^6\)

Finally, we have omitted any consideration to the case in which the fail activity is an alternative of the choice, because this fail action could not ever be executed, so it could be removed. Of course, for the trivial case \( \text{fail} \land \text{fail} \) the translation would be the same as that of fail.

6. Correctness

In this section we show that this translation is correct, in the sense that given a choreography \( C \) that uses a set of natural variables \( \text{Var} \), its operational semantics and the corresponding NTA behave in the same way, by generating the same contextual timed traces, but abstracting from their internal movements.

**Theorem 1.** Let \( A \) be a WS-CDL activity using a set of variables \( \text{Var} \), with the restrictions introduced, and \( t(A) \) its corresponding NTA as defined in the previous section. Then, for any contextual timed trace \( s \) of \( [\overline{A}] \), there is a Var-contextual timed trace \( s' \) of \( t(A) \) such that \( \phi(s) = \phi'(s') \), where \( \phi \) is a function that removes from \( s \) all the internal movements (empty transitions), and \( \phi' \) a function that removes from \( s' \) both the \( \tau \)-movements and the synchronization movements (introduced by the parallel operator translation). Conversely, for any Var-contextual timed trace \( s' \) of \( t(A) \) there is a contextual timed trace \( s \) of \( [\overline{A}] \) such that \( \phi(s) = \phi'(s') \).

Furthermore, \( s \) is a terminal trace of \( [\overline{A}] \) if and only if \( s' \) is also a terminal trace of \( t(A) \).

**Proof.** We use structural induction on \( A \):

- **Base cases:** These are the assign, noaction, inter and fail:
  - For assign\((r, v, n)\), according to the operational semantics, we may have either time elapsing (elapse is true for assign) or the execution of the basic action assign, which changes the context by replacing the value of variable \( v \) by \( n \). (rule Assign in Table 4). The corresponding automaton (assign in Fig. 1) also allows time elapsing, since the edge connecting both nodes is not urgent. Furthermore, the execution of this edge generates the action assign\((r, v, n)\), which changes the value of \( v \) to \( n \). Consequently, for all contextual timed trace \( s \) of assign\((r, v, n)\), \( s \) is also a Var-contextual timed trace in \( t(\text{assign}(r, v, n)) \), and viceversa. It is also immediate to check that terminal traces coincide in both semantics.
  - For the noaction operator the reasoning is analogous to the assign operator.
  - For inter\((r_1, r_2, v_1, v_2, t)\), we may have the following contextual timed traces according to the operational semantics:

\(^6\) If \( M = \infty \) the time-out edge would not appear.
Time cannot elapse if we take a context \( \mu \) for which \( \mu(v_1) = \epsilon \). This case is also possible in the automaton depicted in Fig. 1 for the inter operator, since when \( v_1 = \epsilon \) the urgent edge labelled with this guard is enabled, and therefore no time can elapse. Furthermore, in this case a fail action is executed both in the operational semantics (rule Int1 in Table 4) and in the automaton (urgent edge).

Time can elapse in both semantic models for all contexts \( \mu \) for which \( \mu(v_1) \neq \epsilon \), but up to \( t + 1 \) time units. In the operational semantics, according to the aging function, after \( t + 1 \) time units we reach the term \( [\text{fail}]_{t+1} \), for which no time can elapse, and the fail action must be immediately executed. The same occurs in the corresponding automaton, because of the invariant associated with the initial location, and the urgent edge labelled with the guard \( x = t + 1 \). Notice that the initial value of clock \( x \) must be 0, which is the reason to include \( x \) in the set of clocks that must be reset before starting this activity. Besides, context changes in Var-contextual timed traces only may affect variables, clocks do not change their values. Hence, time can elapse in the automaton up to \( t + 1 \) time units, before executing a fail action, and there is no Var-contextual timed trace in which fail is executed in advance.

According to rule Int2 in Table 4, for all contexts \( \mu \) for which \( \mu(v_1) \neq \epsilon \), the \( \text{inter}(r_1, r_2, v_1, v_2, t) \) action can be executed after any time \( t' \leq t \), which is captured by the term \( \text{dinter}(r_1, r_2, v_1, v_2, t-t') \). This behaviour is also possible in the automaton, since the edge labelled with this action has associated the guard \( x \leq t \land v_1 \neq \epsilon \).

It is an immediate consequence of the previous distinction of cases that terminal traces coincide in both semantics.

- The only possible transition for fail is the immediate execution of the action fail (rule Fail in Table 4). The same occurs in the corresponding automaton in Fig. 1, since the edge leaving the initial location is urgent, and it is also labelled with the action Fail.

**General cases:** we now assume as induction hypothesis that for any activities \( A_1, A_2 \) (fulfilling the introduced restrictions) the contextual timed traces coincide in both semantics up to functions \( \phi, \phi' \) and those that are terminal for one semantics are also terminal with the other semantics. Let us then see the different cases we have:

- **Sequence:** According to the operational semantics (rules Seq in Tables 3 and 5), the contextual timed traces of \( [A_1; A_2]_n \) are obtained as the union of the contextual timed traces of \( [A_1]_n \) and the traces obtained by the concatenation of terminal contextual timed traces of \( [A_1]_n \) with contextual timed traces of \( [A_2]_n \), except in the event of a failure by \( [A_1]_n \), in which case the terminal trace terminates with the fail action, after which the only possible evolution is time elapsing. Then, for any contextual timed trace \( s \) of \( [A_1; A_2]_n \), we may distinguish two cases:
  - \( s \) is a contextual timed trace of \( [A_1]_n \) (which can terminate with a fail action and time elapsing): by the induction hypothesis we obtain a Var-contextual timed trace \( s' \) in \( [A_1]_n \) such that \( \phi(s) = \phi'(s') \). Traces \( s \) and \( s' \) are respectively contextual (and Var-contextual) timed traces of \( A_1; A_2 \) (operational semantics) and the automaton \( t(A_1; A_2) \), which concludes the first part of the proof. If \( s \) is a terminal trace, \( s' \) will be terminal, too. Then, if the final action of \( s \) is fail, \( s' \) will also have fail as final action, so \( s' \) would also be final for \( t(A_1; A_2) \).

  For the converse the reasoning is analogous.

  - \( s = s_1 \cdot s_2 \), where \( s_1 \) is a terminal trace of \( A_1 \) and \( s_2 \) a contextual timed trace of \( A_2 \). By the induction hypothesis we obtain two Var-contextual timed traces \( s'_1 \) for \( t(A_1) \) (which is terminal) and \( s'_2 \) for \( t(A_2) \), for which \( \phi(s_1) = \phi'(s'_1) \) and \( \phi(s_2) = \phi'(s'_2) \). Observe now that \( s'_1 \cdot s'_2 \) is a Var-contextual timed trace of \( t(A_1; A_2) \), because all the clocks that need to be reset before starting \( t(A_2) \) are included in all the edges reaching init\( A_2 \). The terminal traces of \( A_1; A_2 \) are those obtained for terminal traces \( s_2 \) of \( A_2 \). Hence, since \( s'_2 \) will also be terminal in that case, the Var-contextual timed trace \( s'_1 \cdot s'_2 \) would be terminal for \( t(A_1; A_2) \).

  The converse is analogous.

- **\( A_1 || A_2 \):** Its contextual timed traces are obtained by the interleaving of those of \( A_1 \) and \( A_2 \), but delay transitions must be performed by both activities. Then, for all contextual timed trace \( s \) of \( A_1 || A_2 \) it follows that there are two contextual timed traces \( s_1 \) and \( s_2 \), of \( A_1 \) and \( A_2 \), respectively, which perform the same delay transitions. In the corresponding timed automata (Fig. 1), using the induction hypothesis for both traces, we can obtain a corresponding Var-contextual timed trace (up to \( \phi' \), which hides the initial and final synchronizations on channel \( c \)), because the guards of the edges do not change their value by the addition of the conditions \( v_1 = 0 \) as far as no fail transition is executed, and the same occurs for the invariants associated to locations, because the condition \( v_1 = 1 \) will be false until a fail transition is executed. Notice that we are not only considering the timed traces that each activity can generate isolaely, but also those that could be generated if some variables in \( Var \) change their value throughout its evolution. Thus, this trace semantics captures the fact that one of the involved parallel activities may change the value of some variables in \( Var \), thus affecting the later behaviour of the other activity. Of course, this is not the case for the new variable \( v_1 \), which is not used internally in these activities.

Let us now consider the case in which \( s \) terminates in a fail action, i.e., either \( A_1 \) or \( A_2 \) executes a fail transition (let us say \( A_1 \)). In this case, using the induction hypothesis we could obtain a corresponding trace in \( t(A_1) \), which is also a trace of \( t(A_1; A_2) \), which terminates by changing the value of \( v_1 \) to 1, upon the execution of the corresponding fail edge. Time elapsing is possible after the fail action in the operational semantics, which is also the case in the corresponding automata, because of the change introduced in the invariants: the condition \( v_1 = 1 \) is true, which allows unlimited time elapsing in the current location of the automaton that did not fail, which would actually in that location forever. This does not cause any problems, because a fail transition leads us to the error location in the case of the main activity of the choreography (definition of choreography translation, Fig. 3), so there is no
Possibility to reevaluate the parallel activity again, and in the case of the exception activity the double exception location \((de)\) is reached, from which no edge comes out.

Furthermore, when \(s\) is terminal, either \(s_1\) and \(s_2\) are both terminal or one of them terminates with a fail transition. In the first case, the corresponding Var-contextual timed trace of \(t(A_1 \parallel A_2)\) would terminate with the synchronization on channel \(c\), reaching the final location \(end_{A_1 \parallel A_2}\), so it would be a terminal trace of \(t(A_1 \parallel A_2)\). In the second case, as seen before, the “error location” of \(t(A_1 \parallel A_2)\) is reached with the corresponding Var-contextual timed trace, which is therefore a terminal trace, too.

For the converse, let us take a contextual timed trace \(s'\) of \(t(A_1 \parallel A_2)\). According to the automata depicted in Fig. 1 this trace starts with a \(c\) action, resulting from the initial synchronization on channel \(c\), which is hidden by \(\phi'\), after which we may have some actions and delays. Actions are executed by one of the automata, whereas delays are executed simultaneously by both automata. Thus, \(s'\) can be obtained as the interleaving of two contextual timed traces, \(s'_1\) of \(t(A_1)\) and \(s'_2\) of \(t(A_2)\), which perform the same delay transitions. It may occur that some action executed by one of the automata changes a variable value, which of course affects the subsequent behaviour of the other automaton. Nevertheless, since we are considering the Var-contextual timed traces the resulting traces from these variable changes would also be considered, so using the induction hypothesis we would obtain a couple of Var-contextual timed traces, \(s_1, s_2\), for \(A_1\) and \(A_2\), respectively, corresponding to \(s'_1, s'_2\), which perform the same delay transitions. The interleaving of these traces is a Var-contextual timed trace of \(A_1 \parallel A_2\).

Let us now consider the case in which \(s'\) contains a fail action. Without loss of generality we can assume this action to be executed by \((t(A_1)\). According to the translation, the execution of fail edge leads \((t(A_1)\) to the error location (or the double exception location), and \((t(A_2)\) also becomes to a state in which no edge can be executed (due to the change of \(v_e\)), but time elapsing is possible for its current location (also due to the new value of \(v_e\)). Thus, using again the induction hypothesis we can obtain a couple of Var-contextual timed traces for \(A_1\) and \(A_2\) whose interleaving would result again in a Var-contextual timed trace of \(A_1 \parallel A_2\), in which after the fail action only time elapsing is possible.

Finally, notice that terminal traces (those reaching the location \(end_{A_1 \parallel A_2}\)) terminate with a synchronization on channel \(c\) (which is again hidden by \(\phi'\)), and correspond to terminal traces of \(A_1 \parallel A_2\). For these terminal traces the same reasoning that we have used above is valid, but notice that upon the execution of the final synchronization the initial location of \(t(A_2)\) is reached again, which allows the reactivation of the parallel activity (if it appears inside a repetitive workunit).

- **workunit**\((g, block, g', A_1)\): some cases must be distinguished here, depending on the value of \(block\) and the guard evaluation. When \(block\) is true, time can elapse in both semantics until \(g\) evaluates to true, in which case the activity is immediately started (rule Work1 in Table 4). In the operational semantics we then obtain a repetitive concatenation of contextual timed traces of \([A_1]_\tau\), until \(g'\) becomes false (rules Work4 and Work6), in which case a null movement terminates the trace (rule Work7). This null movement is hidden by \(\phi\), thus, after the application of \(\phi\) we will only see the concatenation of contextual timed traces of \([A_1]_\tau\). This behaviour can also be obtained in the automata model (workunit with block True, Fig. 2), using the induction hypothesis and taking into account that \(\phi'\) hides the \(\tau\) actions. Observe that the three edges labelled with \(\tau\) are urgent, so no time can elapse for them when they are enabled. Thus, initially time can only elapse until \(g\) becomes true, once this condition is fulfilled, the automaton \((t(A)\) is started. Then, we have the Var-contextual timed traces of \((t(A)\), and when we reach a terminal trace (which corresponds to a terminal trace of the operational semantics), we immediately evaluate \(g'\), in order to decide if the automaton is restarted or we terminate by reaching the location \(end_{\phi'\tau}\).

On the other hand, when the activity inside the workunit fails (rules Work2 and Work5 in Table 4) the only possible behaviour once \(g\) is satisfied is the execution of fail, thus obtaining a terminal trace. This behaviour is also obtained in the automata model, just applying the induction hypothesis, because control would be transferred to the corresponding error location.

When \(block\) is false, time cannot elapse for both semantics. Thus, if \(g\) is false the workunit is immediately abandoned with a null movement in the operational semantics (rule Work3 in Table 4), which is also the case for the corresponding automata model (workunit with block False, Fig. 2), because in this case we have an urgent \(\tau\)-edge connecting the initial location with the final location. Both the null movement and the \(\tau\) action are hidden by \(\phi\) and \(\phi'\), respectively, so the resulting behaviour is the same. However, when \(g\) is true, the behaviour is the same as before, i.e., a repetitive concatenation of contextual timed traces of \([A_1]_\tau\), which can also be reproduced in the automata model (up to \(\phi, \phi'\)), since this part of the transformation is identical to the case in which block is true.

For the converse, the reasoning is analogous: for every Var-contextual timed trace of the timed automata model we may obtain (by applying the induction hypothesis on each iteration) a corresponding contextual timed trace in the operational semantics. Notice that it is essential for that that all the clocks required by \(t(A)\) are reset before restarting it, and for that purpose the function \(\varphi_2(A, h, l)\) has been included as the set of clocks to be reset in both (urgent) edges that activate \(t(A)\). Besides, terminal traces of \(t(A)\) have their analogous on the operational semantics, hence it is immediate to conclude that terminal traces of the workunit automata model have their analogous (up to \(\phi, \phi'\)) in the operational semantics.

- **Choice**: we use an extended choice with the syntax indicated in Fig. 2. From the operational semantics it follows that any alternative can proceed by executing its first activity (rules Choi1–2 in Table 5). Thus, for all non-failing
contextual timed trace of the choice \( s \), it follows that \( s \) is a non-failing contextual timed trace of one the alternative activities. We can then apply the induction hypothesis to conclude that there exists a corresponding non-failing Var-contextual timed trace \( s' \) of the activity automata model, and that if the first one was terminal, the second is terminal, too. Some cases can now be distinguished, but are quite similar, they correspond to the execution of an assign, a noaction, an interaction or a workunit as first activity of the chosen alternative. The translation for these cases (Fig. 2) corresponds to the translation made for these basic operators in Fig. 1, concatenated with the timed automata of the sequel activity. Hence \( s' \) is also a Var-contextual timed trace of the choice automata model of Fig. 2. Besides, when \( s \) is a terminal trace, \( s' \) will be terminal, too, but the inner automata, however, the execution of the final edge that leads to endc (labelled with \( \tau \) ) is hidden by \( \phi' \), so the obtained Var-contextual timed trace still coincides with \( s \) (up to \( \phi, \phi' \)).

Let us now consider the case of a failing trace in the operational semantics. An initial fail action can only occur when all the alternatives are able to fail (rule Choi3 in Table 5). Hence, taking into account the syntax introduced for the extended choice in Fig. 2, it follows that all the guards \( g_i \) must be true for that, and additionally, either the fail is a consequence of a time-out that has expired, or because all the source variables of the interactions (even for the first interactions of workunits) are undefined. These cases are contemplated by the translation of Fig. 4, which contains the urgent fail edges that must be included in the translation presented in Fig. 2. Notice that \( M \) is defined as the maximum of all the involved time-outs plus one, so the guard condition \( x = M + 1 \) will only be true when all the time-outs have expired.

For non-initial failing actions we can apply the induction hypothesis in order to obtain the corresponding failing Var-contextual timed trace in the automata model (using the same reasoning as above). The reasoning is analogous for the converse. □

**Corollary 1.** Let \( C = (A_1, A_2) \) be a choreography that uses the set of variables Var and \( \mathcal{N} \) the associated NTA. Then, for any contextual timed trace \( s \) of \( C \) there is a Var-contextual timed trace \( s' \) of \( \mathcal{N} \) such that \( \phi(s) = \phi'(s') \). Conversely, for any Var-contextual timed trace \( s' \) of \( \mathcal{N} \) there is a contextual timed trace \( s \) of \( C \) such that \( \phi(s) = \phi'(s') \).

7. Web Services Translation tool

Web Services Translation tool (WST) is an integrated environment for translating UML 2.0 sequence diagrams into WS-CDL specification documents and, in turn, the WS-CDL specifications are translated into Timed Automata, which are then used to simulate and verify the system behaviour. This tool is available at the site http://www.dsi.uclm.es/retics/WST/
the tool and its source code can be downloaded (the XSL files applied for the translations are defined in the folder called XSLFiles). Moreover, we also provide a bunch of examples in the Documentation section.

In both translations, the tool applies several XSL Style sheets to an initial XML document to obtain another XML document. Three XSL Style sheets are applied to translate the XMI document corresponding to the sequence diagram we have modelled into a WS-CDL document. We also use three XSL Style sheets in order to translate a WS-CDL document into another XML document representing a Timed Automata system in a format readable by the UPPAAL tool.

The interface of the WST tool is divided into three different tabs:

- The first tab, called “RT-UML DIAGRAM”, is an editor that allows the user to model a UML sequence diagram representing the interactions between the different parties in a Web Services composition. This diagram can be exported to XMI format, a step required to obtain the WS-CDL specification document.
- The second tab, called “RT-UML2WS-CDL”, allows the user to automatically translate the XMI code generated in the previous tab into a WS-CDL specification document.
- The third tab, called “WS-CDL2TimedAutomata”, allows the user to automatically translate a WS-CDL specification document into a Timed Automata system. The XML document we obtain in this case can be opened by the UPPAAL tool. In Fig. 5 we can see a snapshot of this tab of the tool. On the left-hand side we have the textbox where we load the WS-CDL code we want to translate. Pressing the button called “Transform” the Timed Automata system code is automatically generated and loaded in the textbox on the right-hand side. Finally, there is a button called “UPPAAL” that can be used to directly open the generated Timed Automata system in the UPPAAL tool.

8. Case study: list of registered voters management

The use of Web Services for e-government has become more and more important in the last years. The expression e-government (from electronic government) refers to the use of Internet technology for providing services and exchanging information related to the government administration. The Service Oriented Computing paradigm provides an ideal framework for this kind of interactions.

In this case study we present a Service Oriented System that manages the lists of registered voters in a country. We distinguish two different kinds of lists: the federal lists and the local lists, for general and local elections, respectively. The following restrictions must be taken into account:

1. A European Union citizen (but not a Spanish citizen) living in Spain could vote in the local elections of his city, but cannot vote in the general elections.
2. A Spanish citizen who is living abroad could only vote in the general elections.
3. A Spanish citizen who is living in Spain could vote in both, general and local elections.

We focus on the case of a citizen who decides to register in these lists. In Fig. 6 we show the different parts of our system: the citizen who interacts with the administration, the registry application that allows citizens to access the e-government procedures, the shared repository that contains all the information about the citizens and the communication protocols, and the multiple services for the different federal and local administrations.

When a citizen decides to register in the lists of voters, he has first to login in the system through the registry application. For the sake of simplicity, we are supposing that the login information sent by the user is always valid. After login, the registry application sends the login information to the shared repository, which has to be sent within a space of 5 min at most. Later, all the information about the citizen is extracted from the database of the shared repository, as well as the procedures that the citizen could use. This information and these procedures are sent in parallel to the registry application. Afterwards, the system shows the citizen the possibility of inscribing himself in any possible list of voters, depending on the circumstances listed before.

![Fig. 6. System diagram.](image-url)
8.1. WS-CDL generation phase

In Fig. 7 we show part of the WS-CDL code corresponding to this case study. We only focus on the parts that are involved in the translation into timed automata, omitting the rest of the code.

Fig. 8 depicts the specification of the case study in the algebraic language that we use as a metamodel of WS-CDL. Letters $A$, $C$, and $D$ correspond to the interactions executed in a sequence at the beginning of the choreography. Letters $B_1$ and $B_2$ correspond to the interactions executed in parallel after $A$, but before $C$. Letters $E_1$, $E_2$ and $E_3$ correspond to the options for the different kinds of citizen that can be executing the process. Finally, letters $F$ and $H$ refer to the interaction with a local administration, while letters $G$ and $I$ refer to the interaction with a federal administration. In the case of a Spanish citizen living in Spain, the system first interacts with the local administration ($H$) and after that, it interacts with the federal administration ($I$).

```xml
...<sequence>
  <interaction name="Login of the citizen in the system"
    channelVariable="RegistryRepositoryChannel"
    operation="LoginInSystem">
    <description type="description">
      Sending the citizen user and password to Repository
    </description>
    <participate relationshipType="RegistryRepository"
      fromRole="Registry Application"
      toRole="Shared Repository"/>
    <exchange name="LoginUser"
      action="request">
      <use variable="Login"/>
      <populate variable="User"/>
    </exchange>
  </interaction>
</sequence>
  ...

<parallel>
  <interaction name="Information about the citizen from Repository"
    channelVariable="RegistryRepositoryChannel"
    operation="CitizenInfoFromRepository">
    <description type="description">
      Sending the citizen information from Repository
    </description>
    <participate relationshipType="RegistryRepository"
      fromRole="Shared Repository"
      toRole="Registry Application"/>
    <exchange name="UserDataCitizenInfo"
      action="response">
      <use variable="UserData"/>
      <populate variable="CitizenInfo"/>
    </exchange>
  </interaction>
  ...

<interaction name="Information about procedures from Repository"
  channelVariable="RegistryRepositoryChannel"
  operation="ProceduresInfoFromRepository">
  <description type="description">
    Sending the procedures information from Repository
  </description>
  <participate relationshipType="RegistryRepository"
    fromRole="Shared Repository"
    toRole="Registry Application"/>
  <exchange name="ProceduresDataProceduresInfo"
    action="response">
    <use variable="ProceduresData"/>
    <populate variable="ProceduresInfo"/>
  </exchange>
  </interaction>
</parallel>
...
```

Fig. 7. WS-CDL specification for the case study.
Fig. 8. Algebraic specification of the case study.

\[
\text{RegisteredVoters} = A_1 \parallel B_2; \parallel C; \parallel D: (E_1 \parallel E_2 \parallel E_3)
\]

\[
A = \text{inter}(\text{Registry, Repository, Login, User, } \infty)
\]

\[
B_1 = \text{inter}(\text{Repository, Registry, User Data, CitizenInfo, } \infty)
\]

\[
B_2 = \text{inter}(\text{Repository, Registry, ProceduresData, ProceduresInfo, } \infty)
\]

\[
C = \text{inter}(\text{Registry, Repository, SelOption, Option, } \infty)
\]

\[
D = \text{inter}(\text{Repository, Registry, List, ListInfo, } \infty)
\]

\[
E_1 = \text{workunit}(\text{CitizenType} = \text{EuropeanInSpain, true, false, } F)
\]

\[
E_2 = \text{workunit}(\text{CitizenType} = \text{SpanishOutSpain, true, false, } G)
\]

\[
E_3 = \text{workunit}(\text{CitizenType} = \text{SpanishInSpain, true, false, } H; I)
\]

\[
F, H = \text{inter}(\text{Registry, Local, CitizenInfo, Citizen, } \infty)
\]

\[
G, I = \text{inter}(\text{Registry, Federal, CitizenInfo, Citizen, } \infty)
\]

Fig. 9. Timed automata for the case study.
Fig. 9 shows the translation into timed automata by applying the rules described in Section 5. We can distinguish two automata here: the Main automaton corresponding to the whole choreography and the Parallel automaton that implements the parallel interaction $B_2$.

8.2. Validation and verification

Once we have obtained the timed automata, we use the UPPAAL tool to check the properties of interest in our system, which are the following:

1) **Information Sending On Time.** We want to see if the system reaches the exception location when a time-out occurs, i.e., when the citizen spends more than 5 min doing nothing after login correctly. The query is specified in the following way in UPPAAL:

$$ (\text{Main.Init}\_\text{Session} \land x > 5) \rightarrow \text{Main.Exception}$$

where Init\_Session is the name of the location previous to the execution of the interaction C. We obtain that this formula is satisfied.

2) **European In Session E.** We want to prove that the registry application finally interacts with a local administration when the citizen is an European living in Spain, that is, interaction $F$ is executed. We call After\_F the location just after executing interaction $F$ and we assign code number 1 to an European citizen. Then, the query is specified as follows:

$$ \text{A[\text{Main.After}_F] imply \text{Main.CitizenType} == 1}$$

We obtain that this formula is satisfied.

3) **Spanish Abroad in Session SA.** We want to prove that the registry application interacts with a federal administration when the citizen is a Spaniard living outside Spain, so interaction $G$ is executed. We call After\_G the location just after executing interaction $G$ and code number 2 corresponds to a Spanish citizen living abroad. Now we have the following query:

$$ \text{A[\text{Main.After}_G] imply \text{Main.CitizenType} == 2}$$

We obtain that this formula is satisfied.

4) **Spanish in Spain in Session SS.** We want to prove that the registry application interacts with a local and a federal administration when the citizen is a Spanish living in Spain, that is, interactions $H$ and $I$ are executed. We call After\_I the location after executing interactions $H$ and $I$, and code number 3 corresponds to a Spanish citizen living in Spain. In this case we have the following query:

$$ \text{A[\text{Main.After}_I] imply \text{Main.CitizenType} == 3}$$

We obtain that this formula is satisfied.

5) **Ends On Time.** Finally, we want to prove that the process finishes in 6 min at most, i.e., interaction $F$, interaction $G$ or interactions $H$ and $I$ are executed in 6 min after login. Otherwise, an exception will be raised. We call Before\_Choice the location before executing any of this interactions, so the query in UPPAAL for this property is:

$$ (\text{Main.Before}_\text{Choice} \land x > 6) \rightarrow \text{Main.Exception}$$

In this case, we obtain that the formula is not satisfied.
At this point, we have to go back to the WS-CDL generation phase and modify the specification to fulfil the last property. The solution is adding a time-out of 6 min to interactions $F$ and $H$ (“Send Inscription to Local Admin”) and interactions $G$ and $I$ (“Send Inscription to Federal Admin”), and also to interaction $D$ (“Information about the corresponding list”). This modification guarantees that it is not possible to finish the process later than 6 min without raising an exception.

Fig. 10 shows the modifications in the **Main** automaton corresponding to the new specification. We can see that new invariants and guards are added corresponding to the new time-outs.

Lastly, we check again the five properties described before with the UPPAAL tool. Now, we obtain that the automata do **satisfy** all the requirements.

### 9. Conclusions and future work

WS-CDL (Web Services Choreography Description Language) is a W3C proposal for the description of Web Services choreographies. The choreographic viewpoint of a composite Web service aims at describing the collaborations between the involved partners regardless of the supporting platform or programming model used by the implementation of the hosting environment. WS-CDL therefore includes a repertoire of activity constructions that capture the relationship between the actors involved in the choreography.

In this paper we have defined an algebraic language with a syntax inspired in that of WS-CDL, in which its more relevant activity constructions have been considered, and a barred operational semantics has been defined for it. One important aspect of this algebraic language is that we have paid special attention to the timing aspects of WS-CDL. Furthermore, we have also defined a translation of WS-CDL specifications into a network of timed automata, showing the benefits of this translation, as the possibility of simulate, validate and verify some properties of the described system, by using a tool supporting the NTA model, such as the UPPAAL tool.

One of the main contributions of this paper is the formalization of WS-CDL semantics, which is presented in a textual way in [23], with the result that this ‘official semantics’ suffers from many deficiencies and ambiguities, which are solvable with a formal semantics. Furthermore, the use of a very well known formalism, such as timed automata, in order to obtain an alternative representation of the system is another important contribution of this paper, since this alternative representation can be used to analyse the system behaviour systematically.

As future work we plan to expand the subset of WS-CDL by considering some additional features, like the hierarchy of choreographies and finalizer blocks. The extension to a hierarchy of choreographies requires an extension of the syntax of our meta-model, by considering the **perform** activity of WS-CDL, which is used to invoke an inner choreography; whereas finalizer blocks should be included as a third component in choreographies, and **finalize activities** of WS-CDL should also be considered in order to activate these finalizer blocks.

Other issue that we plan to deal with is the conformance of orchestrations with respect to choreographies, which has already been studied by van der Alst et al. [28], Honda et al. [18] and Bravetti and Zavattaro [6, 7]. However there are still some problems to solve; for instance, [13] states the problem of automatically deriving choreography-conforming systems of services. Conformance is defined in terms of whether a set of processes behaves as it is stated by a specification. Therefore, orchestrations play the role of processes and the starting specification is usually a choreography. We will deal with this problem from two different viewpoints. Choreographies describe communication patterns among the participants, so it is possible to extract the communication pattern for each participant. These patterns, therefore, can be used to generate orchestration skeletons that by construction will conform to the given choreography. The other approach we plan to investigate takes as input both a choreography and an orchestration, and we plan to define some techniques that allow us to establish whether conformance holds or not when time restrictions are taken into account.

### References

2. A. Arkin et al., Web Service Choreography Interface (WSCI) 1.0. <http://www.w3.org/TR/wsci/>.


