Web Services Choreographies Verification ⋆

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.

Key words: Web Services, WS-CDL, Verification, Choreography, UPPAAL, Timed Automata, Simulation.

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, as WS-CDL (Web Services Choreography Description Language) [17], WSCI (Web Service Choreography Interface) [2] or DAML-S [1,18]. With these

⋆ Supported by the Spanish government (co-financed by FEDER founds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PAC06-0008-6995.

Email addresses: MEmilia.Cambronero@uclm.es (M. Emilia Cambronero), Gregorio.Diaz@uclm.es (Gregorio Díaz), Valentin.Valero@uclm.es (Valentín Valero), emartinez@dsi.uclm.es (Enrique Martínez).

languages, the collaborations and the conditions under which these collaborations 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, then, 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 one specific services perspective. WS-CDL, as an standards exemplar, 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 party 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 exceptional execution) and prescribing the rules for consistently managing their non-observable data. Thus, the orchestration refers to 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 [12]. This paper, then, 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 UPAAL tool [13], 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 most important 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. Afterwards, the translation is formally defined along with the proof of soundness, in the sense that both the operational semantics of a term of the meta-model

and the corresponding network of timed automata behave in the same way.

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. 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. A Case Study that illustrates the translation is presented in Section 7, and finally, the conclusions and future work are presented in Section 8.

2 Related Work

The developers of WS-CDL claim that its design has been based on a formal language, the π-calculus [16], 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 [7], where the authors compare a formalized version of WS-CDL, called global calculus, with the π-calculus. They discuss how the same business protocols can be described by WS-CDL and π-calculus equivalently, as two different ways of describing communication-centered software in the form of formal calculi.

Jin Song Dong et al. [9] 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, then, is that they work with orchestrations, rather than choreographies. Thus, *Orc* is a language close to WS-BPEL. Howard Foster et al. [11] 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 produced by using Finite State Process (FSP) as an intermediate language. One of the most important contributions of the group leaded 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. [22], 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 [8] 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: Xu tao Du et al [10] 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. N. Sharygina [20] presents a model checking with abstraction technique 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 [21] a timed-prioritized Petri net model is used to represent the behaviour of composite Web Services described by a prioritized version of WS-CDL. Lohman et al. [14] use open workflow net models and define a fully-automatic translation of this formalism into abstract BPEL by using the Tools4BPEL framework.

There are also translations that use algebraic models: Salaun et al. [19] have defined a process algebra to derive the interactive behaviour of a business process starting from a BPEL4WS specification, A. Brogi et al. [6] have defined a translation of WSCI (Web Service Choreography Interface) to CCS [15], showing the benefits of such translation, and W.L. Yeung [23] has defined a mapping from WS-CDL and BPEL4WS into CSP, providing a formal approach to verifying the behaviour of collaborating Web services.

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 [17] basically consists of parties, roles, the exchanged information description, choreographies, channels and activities. Parties 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 parties 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 plain WS-CDL documents, 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.

Channels are used to establish where and how the information is exchanged, but they are unimportant for our purposes, so we will abstract them in our meta-model. The collaborative behaviour of the participants in a choreography is described by means of *activities*. These are the lowest level components of 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 a 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 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.

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 *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[1], 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, then, for the activities is defined by the following BNF-notation:

$$A ::= \mathit{fail} \mid \mathit{assign}(r, v, n) \mid \mathit{noaction}(r) \mid \mathit{inter}(r_1, r_2, v_1, v_2, t) \mid A\,;A \mid$$
$$A \,\Box\, A \mid A \parallel A \mid \mathit{workunit}(g, \mathit{block}, g', A)$$

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 Var, $n \in \mathbb{N}$, g, g' are predicates that use the variable names in Var, and *block* is a boolean. Given a predicate g, we will call $\mathit{Vars}(g)$ the set of variables used in g.

The basic activities are *fail*, *assign*, *noaction* and *inter*; *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, then, 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, parallel and workunit. The workunit operator has 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, then, 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

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

of those enabled in the choice [2] 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.

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 $dinter(r_1, r_2, v_1, v_2, t, t')$, 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 expiration. 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:

$$D ::= \overline{B} \quad | \quad \underline{B} \quad | \quad D\,;B \quad | \quad B\,;D \quad | \quad D \,\Box\, B \quad | \quad B \,\Box D \quad |$$
$$D \parallel D \quad | \quad workunit(g, block, g', D)$$

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 $A = workunit(g, true, g', assign(r, v, 1))$. 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, then, we reach the dynamic term $D_1 = workunit(g, true, g', \overline{assign(r, v, 1)})$, 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: $D_2 = workunit(g, true, g', \underline{assign(r, v, 1)})$, from which g' is evaluated. If some variable needed to evaluate g' is not available or g' is false, then, the workunit ends and the dynamic term $\underline{A}$ is reached. Otherwise, when g' is true, D_1 is reached again.

□

[2] In the sense that it can execute some action at the current instant.

(Seq1)	$\overline{B_1; B_2} \equiv \overline{B_1}; B_2$	**(Seq2)**	$\underline{B_1}; B_2 \equiv B_1; \overline{B_2}$
(Seq3)	$B_1; \underline{B_2} \equiv \underline{B_1; B_2}$	**(Cho1)**	$\overline{B_1 \,\Box\, B_2} \equiv \overline{B_1} \,\Box\, B_2$
(Cho2)	$\overline{B_1 \,\Box\, B_2} \equiv B_1 \,\Box\, \overline{B_2}$	**(Cho3)**	$\underline{B_1} \,\Box\, B_2 \equiv \underline{B_1 \,\Box\, B_2}$
(Cho4)	$B_1 \,\Box\, \underline{B_2} \equiv \underline{B_1 \,\Box\, B_2}$	**(Par1)**	$\overline{B_1 \parallel B_2} \equiv \overline{B_1} \parallel \overline{B_2}$
(Par2)	$\underline{B_1} \parallel \underline{B_2} \equiv \underline{B_1 \parallel B_2}$		
(Inter)	$\overline{inter(r_1, r_2, v_1, v_2, t)} \equiv \overline{dinter(r_1, r_2, v_1, v_2, t, t)}$		
(Cong1)	$\dfrac{\forall op \in \{;, \Box\}, \quad D_1 \equiv D_2}{B \; op \; D_1 \equiv B \; op \; D_2\,, \quad D_1 \; op \; B \equiv D_2 \; op \; B}$		
(Cong2)	$\dfrac{D_1 \equiv D_2}{D \parallel D_1 \equiv D \parallel D_2\,, \quad D_1 \parallel D \equiv D_2 \parallel D}$		
(Cong3)	$\dfrac{D_1 \equiv D_2}{workunit(g, block, g', D_1) \equiv workunit(g, block, g', D_2)}$		

Table 1
Equivalence rules

In this example we have used dynamic terms to represent the current state of the system. However, dynamic terms like $\overline{B_1 \,\Box\, B_2}$, $\overline{B_1} \,\Box\, B_2$ and $B_1 \,\Box\, \overline{B_2}$ 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 1. By means of this equivalence relation we can identify those dynamic terms that can be obtained by movingthe 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 *CDterms*.

The rules of Table 1 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. *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 $\mathit{init}(D)$ (resp. $\mathit{final}(D)$), when there exists an extended activity B such that $\overline{B} \in [D]_{\equiv}$ (resp. $\underline{B} \in [D]_{\equiv}$). In such a case we will say that the class $[D]_{\equiv}$ is initial (resp. final) too.

□

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

A choreography is executed within the context of the variables defined in it, where a context μ is defined as a function $\mu : \mathit{Var} \to \mathbb{N} \cup \{\epsilon\}$, which assigns a value to every variable, where unavailable variables are assigned the ϵ value. We denote the set of possible contexts of a choreography by *Contexts*. The *initial context*, denoted by μ_0, is that defined by assigning ϵ to all the variables in the choreography, except the *clock*, which is assigned to 0. Given a context μ, a variable v and an integer arithmetic expression n, we denote by $\mu[v/n]$ the context obtained from μ by changing the value of v to n. Given a predicate g and a context μ, we will write $\mathit{sat}(\mu, g)$ when $\forall v \in \mathit{Vars}(g)$, $\mu(v) \neq \epsilon$, and g evaluates to true under μ.

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 $\mathit{aging} : \mathit{CDterms} \to \mathit{CDterms}$ is defined in a structural way, as follows:

For any dynamic terms D, D_1, D_2:

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

(9) If $\neg final(D) \;:\; aging([D;B]_{\equiv}) = [D';B]_{\equiv}$, and $aging([B;D]_{\equiv}) = [B;D']_{\equiv}$, with $D' \in aging([D]_{\equiv})$.

(10) If $\neg init(D) \wedge \neg final(D) \;:\; aging([B \,\Box\, D]_{\equiv}) = [B \,\Box\, D']_{\equiv}$, and $aging([D \,\Box\, B]_{\equiv}) = [D' \,\Box\, B]_{\equiv}$, with $D' \in aging([D]_{\equiv})$.

(11) If $init(D) \wedge \neg final(D) \;:\; aging([B \,\Box\, D]_{\equiv}) = [B' \,\Box\, D']_{\equiv}$, and $aging([D \,\Box\, B]_{\equiv}) = [D' \,\Box\, B']_{\equiv}$, with $D' \in aging([D]_{\equiv})$ and B' such that $\overline{B'} \in aging([\overline{B}]_{\equiv})$.

(12) If $\neg final(D_1) \wedge \neg final(D_2) : aging([D_1 \parallel D_2]_{\equiv}) = [D_1' \parallel D_2']_{\equiv}$, with $D_1' \in aging([D_1]_{\equiv})$ and $D_2' \in aging([D_2]_{\equiv})$.

(13) If $final(D_1) \wedge \neg final(D_2) : aging([D_1 \parallel D_2]_{\equiv}) = [D_1 \parallel D_2']_{\equiv}$, and $aging([D_2 \parallel D_1]_{\equiv}) = [D_2' \parallel D_1]_{\equiv}$, with $D_2' \in aging([D_2]_{\equiv})$. □

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 $([D]_{\equiv}, \mu)$, where D is a dynamic term and μ a context. Then, 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 $elapse : CDterms \times Contexts \rightarrow Boolean$ is defined in a structural way, as follows:

For any dynamic terms D, D_1, D_2 and any context μ:

(1) If $final(D)$, then $elapse([D]_{\equiv}, \mu) = true$.

(2) $elapse([\overline{fail}]_{\equiv}, \mu) = false$.

(3) $elapse([\overline{assign(r,v,n)}]_{\equiv}, \mu) = true$.

(4) $elapse([\overline{noaction(r)}]_{\equiv}, \mu) = true$.

(5) If $\mu(v_1) \neq \epsilon$: $elapse([\overline{dinter(r_1, r_2, v_1, v_2, t, t')}]_{\equiv}, \mu) = true$.

(6) If $\mu(v_1) = \epsilon$: $elapse([\overline{dinter(r_1, r_2, v_1, v_2, t, t')}]_{\equiv}, \mu) = false$.

(7) $elapse([\overline{workunit(g, block, g', B)}]_{\equiv}, \mu) = block$.
(8) If $\neg final(D)$: $elapse([workunit(g, block, g', D)]_{\equiv}, \mu) = elapse([D]_{\equiv})$.
(9) If $final(D)$: $elapse([workunit(g, block, g', D)]_{\equiv}, \mu) = false$.
(10) If $\neg final(D)$:
$elapse([D; B]_{\equiv}, \mu) = elapse([B; D]_{\equiv}, \mu) = elapse([D]_{\equiv}, \mu)$.
(11) If $\neg init(D) \wedge \neg final(D)$:
$elapse([B \,\Box\, D]_{\equiv}, \mu) = elapse([D \,\Box\, B]_{\equiv}, \mu) = elapse([D]_{\equiv}, \mu)$.
(12) If $init(D)$:
$elapse([B \,\Box\, D]_{\equiv}, \mu) = elapse([D \,\Box\, B]_{\equiv}, \mu) =$
$elapse([\overline{B}]_{\equiv}, \mu) \vee elapse([D]_{\equiv}, \mu)$.
(13) If $\neg final(D_1) \wedge \neg final(D_2)$:
$elapse([D_1 \parallel D_2]_{\equiv}, \mu) = elapse([D_1]_{\equiv}, \mu) \wedge elapse([D_2]_{\equiv}, \mu)$.
(14) If $final(D_1) \wedge \neg final(D_2)$:
$elapse([D_1 \parallel D_2]_{\equiv}, \mu) = elapse([D_2 \parallel D_1]_{\equiv}, \mu) = elapse([D_2]_{\equiv}, \mu)$. □

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 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: $([D]_{\equiv}, A_2)$ or $(A_1, [D]_{\equiv})$, where $[D]_{\equiv}$ corresponds to the activity in execution in the choreography (the life-line or its exception block), and A_2 can be empty.

We also define a *contextual dynamic choreography term*, as a pair $(\mathcal{C}, \mu)$, where $\mathcal{C}$ is a dynamic choreography term and μ is a context.

Given a choreography $C = (A_1, A_2)$, the *initial contextual dynamic term* of C is[3] $([\overline{A_1}]_{\equiv}, A_2, \mu_0)$. □

In Tables 2 and 3, we introduce the rules that define the transitions for the contextual activity terms, where we can see that we have two types of transi-

[3] We will write contextual dynamic choreography terms as triples, by omitting the parentheses for the dynamic choreography term.

(Clock) $$\frac{elapse([D]_{\equiv},\mu)}{([D]_{\equiv},\mu) \longrightarrow_1 (aging([D]_{\equiv}),\mu[clock/clock+1])}$$

(Fail) $$\frac{}{(\overline{[\underline{fail}]}_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}$$

(Assign) $$\frac{}{(\overline{[\underline{assign(r,v,n)}]}_{\equiv},\mu) \xrightarrow{assign(r,v,n)} ([\underline{assign(r,v,n)}]_{\equiv},\mu[v/n])}$$

(Noact) $$\frac{}{(\overline{[\underline{noaction(r)}]}_{\equiv},\mu) \xrightarrow{noaction(r)} ([\underline{noaction(r)}]_{\equiv},\mu)}$$

(Int1) $$\frac{\mu(v_1) \neq \epsilon}{(\overline{[\underline{dinter(r_1,r_2,v_1,v_2,t,t')}]}_{\equiv},\mu) \xrightarrow{inter(r_1,r_2,v_1,v_2,t)} ([\underline{dinter(r_1,r_2,v_1,v_2,t,t')}]_{\equiv},\mu[v_2/v_1])}$$

(Int2) $$\frac{\mu(v_1) = \epsilon}{(\overline{[\underline{dinter(r_1,r_2,v_1,v_2,t,t')}]}_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}$$

(Work1) $$\frac{sat(\mu,g),\ ([\overline{B}]_{\equiv},\mu) \xrightarrow{a} ([D]_{\equiv},\mu'),\ a \neq fail}{(\overline{[workunit(g,block,g',B)]}_{\equiv},\mu) \xrightarrow{a} (workunit(g,block,g',[D]_{\equiv}),\mu')}$$

(Work2) $$\frac{sat(\mu,g),\ ([\overline{B}]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}{(\overline{[workunit(g,block,g',B)]}_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv}),\mu)}$$

(Work3) $$\frac{\neg sat(\mu,g)}{(\overline{[workunit(g,false,g',B)]}_{\equiv},\mu) \xrightarrow{\emptyset} ([\underline{workunit(g,false,g',B)}]_{\equiv},\mu)}$$

(Work4) $$\frac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu'),\ a \neq fail}{([workunit(g,block,g',D)]_{\equiv},\mu) \xrightarrow{a} ([workunit(g,block,g',D')]_{\equiv},\mu')}$$

(Work5) $$\frac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}{([workunit(g,block,g',D)]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}$$

(Work6) $$\frac{sat(\mu,g'),\ D \equiv \underline{B}}{([workunit(g,block,g',D)]_{\equiv},\mu) \xrightarrow{\emptyset} ([workunit(g,block,g',\overline{B})]_{\equiv},\mu)}$$

(Work7) $$\frac{\neg sat(\mu,g'),\ D \equiv \underline{B}}{([workunit(g,block,g',D)]_{\equiv},\mu) \xrightarrow{\emptyset} ([\underline{workunit(g,block,g',B)}]_{\equiv},\mu)}$$

Table 2
Transition rules for contextual activity terms (I)

tion:

- $([D]_{\equiv},\mu) \longrightarrow_1 ([D']_{\equiv},\mu')$: which represents the passage of one time unit.
- $([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')$: 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 3 we use the notation $([D]_{\equiv},\mu) \xrightarrow{fail} \not\rightarrow$ to mean that no transition labelled with *fail* can be executed from $([D]_{\equiv},\mu)$. Rule *Clock* is used to capture the passage of one time unit. Rules *Fail*, *Assign*

(Seq1-2) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ a \neq \mathit{fail}}{([D;B]_{\equiv},\mu) \xrightarrow{a} ([D';B]_{\equiv},\mu')} \qquad \dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ a \neq \mathit{fail}}{([B;D]_{\equiv},\mu) \xrightarrow{a} ([B;D']_{\equiv},\mu')}$

(Seq3-4) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)\,,}{([D;B]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)} \qquad \dfrac{([D]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)\,,}{([B;D]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)}$

(Choi1-2) $\dfrac{([\overline{B_1}]_{\equiv},\mu) \xrightarrow{a} ([D]_{\equiv},\mu')\,,\ a \neq \mathit{fail}}{([\overline{B_1 \,\Box\, B_2}]_{\equiv},\mu) \xrightarrow{a} ([D \,\Box B_2]_{\equiv},\mu')} \qquad \dfrac{([\overline{B_2}]_{\equiv},\mu) \xrightarrow{a} ([D]_{\equiv},\mu')\,,\ a \neq \mathit{fail}}{([\overline{B_1 \,\Box\, B_2}]_{\equiv},\mu) \xrightarrow{a} ([B_1 \,\Box D]_{\equiv},\mu')}$

(Choi3) $\dfrac{([\overline{B_1}]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)\,,\ ([\overline{B_2}]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)}{([\overline{B_1 \,\Box\, B_2}]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)}$

(Choi4) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ \neg \mathit{init}(D)\,,\ a \neq \mathit{fail}}{([D \,\Box\, B]_{\equiv},\mu) \xrightarrow{a} ([D' \,\Box B]_{\equiv},\mu')}$

(Choi5) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ \neg \mathit{init}(D)\,,\ a \neq \mathit{fail}}{([B \,\Box\, D]_{\equiv},\mu) \xrightarrow{a} ([B \,\Box D']_{\equiv},\mu')}$

(Choi6-7) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)\,,\ \neg \mathit{init}(D)}{([B \,\Box\, D]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)} \qquad \dfrac{([D]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)\,,\ \neg \mathit{init}(D)}{([D \,\Box\, B]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)}$

(Par1) $\dfrac{([D_1]_{\equiv},\mu) \xrightarrow{a} ([D_1']_{\equiv},\mu')\,,\ a \neq \mathit{fail}\,,\ ([D_2]_{\equiv},\mu) \xrightarrow{\mathit{fail}}\!\!\!\!\!/\ }{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{a} ([D_1' \| D_2]_{\equiv},\mu')}$

(Par2) $\dfrac{([D_2]_{\equiv},\mu) \xrightarrow{a} ([D_2']_{\equiv},\mu')\,,\ a \neq \mathit{fail}\,,\ ([D_1]_{\equiv},\mu) \xrightarrow{\mathit{fail}}\!\!\!\!\!/\ }{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{a} ([D_1 \| D_2']_{\equiv},\mu')}$

(Par3-4) $\dfrac{([D_2]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)}{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)} \qquad \dfrac{([D_1]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)}{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{\mathit{fail}} ([\underline{\mathit{fail}}]_{\equiv},\mu)}$

Table 3
Transition rules for contextual activity terms (II)

and *Noact* are evident, whereas *Int1* captures the execution of an activated interaction, when the source variable has a value assigned. Otherwise, rule *Int2* is used to raise an exception. Rules *Work1* to *Work7* establish the semantics of workunits, according to the interpretation described previously. Rules *Seq1* to *Seq4* capture the semantics of the sequence operator, while *Choi1* to *Choi7* define the semantics of the choice. In rule *Choi3* we can see that in a choice we can only execute a *fail* movement 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.

(Cor1)	$\dfrac{([D]_{\equiv},\mu) \longrightarrow_1 ([D']_{\equiv},\mu')}{([D]_{\equiv},A_2,\mu) \longrightarrow_1 ([D']_{\equiv},A_2,\mu')}$	**(Cor2)**	$\dfrac{([D]_{\equiv},\mu) \longrightarrow_1 ([D']_{\equiv},\mu')}{(A_1,[D]_{\equiv},\mu) \longrightarrow_1 (A_1,[D']_{\equiv},\mu')}$
(Cor3)	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu'),\ a \neq fail}{([D]_{\equiv},A_2,\mu) \xrightarrow{a} ([D']_{\equiv},A_2,\mu')}$	**(Cor4)**	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu'),\ a \neq fail}{(A_1,[D]_{\equiv},\mu) \xrightarrow{a} (A_1,[D']_{\equiv},\mu')}$
(Cor5)	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu),\ A_2 \neq \emptyset}{([D]_{\equiv},A_2,\mu) \xrightarrow{fail} (B_1,[\overline{A_2}]_{\equiv},\mu)}$	**(Cor6)**	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}{(A_1,[D]_{\equiv},\mu) \xrightarrow{fail} (A_1,[\underline{fail}]_{\equiv},\mu)}$

Table 4
Transition rules for choreographies

The rules for choreographies are those introduced in Table 4, 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 $q_0 = ([\overline{A_1}]_{\equiv}, A_2, \mu_0)$, 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 $s, s' \in (\mathbb{N} \cup Act)^*$, 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 throughout its evolution.

Definition 5 (Contextual Timed Traces)
Let D be a dynamic activity term. We define the set of *contextual timed traces* of $[D]_{\equiv}$ as follows:

$$Ctr([D]_{\equiv}) = \{\epsilon\} \cup \{s \in (\mathbb{N} \cup Act)^* \mid ([D]_{\equiv},\mu) \xrightarrow{s_1} ([D_1]_{\equiv},\mu_1), \text{ and for all } i \in \{1,\ldots,length(s)\},\ ([D_i]_{\equiv},\mu_i') \xrightarrow{s_{i+1}} ([D_{i+1}]_{\equiv},\mu_{i+1}), \text{ for any contexts } \mu,\mu_i,\mu_i'\}$$

where ϵ stands for an empty trace, s_i is the i^{th}-component of s, $s_i \in \mathbb{N} \cup Act$, $([D_i]_{\equiv},\mu_i) \xrightarrow{s_{i+1}} ([D_{i+1}]_{\equiv},\mu_{i+1})$ denotes the evolution from $([D_i]_{\equiv},\mu_i)$ to $([D_{i+1}]_{\equiv},\mu_{i+1})$, either by executing an action $s_i \in Act$ (possibly empty), or by time elapsing (s_i transitions $\longrightarrow_1$).

□

This definition can be extended to choreographies in a straightforward way. Notice that in this definition we do not only consider the timed traces that are reachable starting from a specific contextual activity term $([D]_{\equiv},\mu)$, but all the timed traces generated considering any intermediate context throughout the evolution of $[D]_{\equiv}$, i.e., for any reachable dynamic activity term D_i we take this term with any possible context and we then consider all its possible evolutions.

4 Timed Automata

A timed safety automaton, or simply timed automaton (TA) [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 transitions, 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.

Definition 6 (Timed Automaton)
We consider a finite set of real-valued variables $\mathcal{C}$ ranged over by $x, y, \ldots$ standing for clocks, a finite set of non-negative integer-valued variables $\mathcal{V}$, ranged over by $v, w, \ldots$ and a finite alphabet Σ ranged over by $a, b, \ldots$ standing for actions. We will use letters $r, r', \ldots$ to denote sets of clocks. We will denote by Ass the set of possible assignments, $Ass = \{v := expr \mid v \in \mathcal{V}\}$, where $expr$ 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: $x \sim n$, $x - y \sim n$, $v \sim n$ or $v - w \sim n$, for $x, y \in \mathcal{C}$, $v, w \in \mathcal{V}$, $\sim \in \{\leq, <, =, >, \geq\}$ 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, $E \subseteq N \times \mathcal{G} \times \Sigma \times \mathcal{P}(Ass) \times 2^{\mathcal{C}} \times N$ 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. $I : N \to \mathcal{G}$ is a function that assigns invariant conditions (which could be empty) to locations.

We will write $n \xrightarrow{g,a,r}_{s} n'$ to denote $(n, g, a, s, r, n') \in E$, and $n \xrightarrow{g,a,r}_{s\,u} n'$ when $(n, g, a, s, r, n') \in E_u$. □

The semantics of a timed automaton is defined as a state transition system, where each state represents a location and a clock and variable valuation. Letters $u, u', \ldots$ will represent clock and variable valuations, $u, u' \in \mathbb{R}_0^{+\mathcal{C}} \times \mathbb{N}^{\mathcal{V}}$. By $u \in g$ we will represent that the valuation u makes g to be true, where we assume that when g is empty $u \in g$ is true. Furthermore, we also assume that g is false when a variable used in g has not been assigned a value. By $u\{s\}$ we represent the valuation obtained from u by changing the value of the variables as indicated by the set of assignments s (which can be empty). $u|_r$ represents the valuation obtained from u by resetting to zero all the clocks in r, and $u + d$ represents the valuation that takes u and increases the value of every clock by d, but keeping in both cases the value of the integer variables.

Definition 7 (Timed Automaton Semantics)
Let $\mathcal{A} = (N, n_0, E, I)$ be a timed automaton. The semantics of $\mathcal{A}$ is defined as the timed labelled transition system $(Q, q_0, \rightarrow)$, where:

- $Q \subseteq N \times (\mathbb{R}_0^{+\mathcal{C}} \times \mathbb{N}^{\mathcal{V}})$ (set of states).
- $q_0 = (n_0, \overline{0}) \in Q$, is the initial state, where $\overline{0}$ is the valuation that assigns every clock to zero and every integer variable to ϵ (a special natural value representing uninitialized variables).
- $\rightarrow \subseteq (Q \times \mathbb{R}_0^+ \times \mathrm{Q}) \cup (\mathrm{Q} \times \Sigma \times Ass \times \mathrm{Q})$ (delay and action transitions).

Delay transitions are in the form (q, d, q'), for $d \in \mathbb{R}_0^+$, denoted by $q \xrightarrow{d} q'$, and are defined by the following rule:

- $(n, u) \xrightarrow{d} (n, u + d)$ if and only if $(u + d') \in I(n)$, for all $d' \leq d$, $d' \in \mathbb{R}_0^+$.

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

- $(n, u) \xrightarrow[s]{a} (n', u')$ if and only if there is an edge $n \xrightarrow[s]{g,a,r} n'$, such that $u \in g$, $u' = (u\{s\})|_r$, and $u' \in I(n')$.

□

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 $a, b \ldots$, whereas synchronization actions must be executed simultaneously by two automata. 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 Def. 7.

Definition 8 (Semantics of an NTA)
Let $\mathcal{A}_i = (N_i, n_{0,i}, E_i, I_i)$, $i = 1, \ldots, k$ be a set of timed automata. A *state* of the NTA $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ is a pair $(\overline{n}, u)$, where $\overline{n} = (n_1, \ldots, n_k)$, 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:

- $(\overline{n}, u) \xrightarrow{d} (\overline{n}, u + d)$ (delay rule) if and only if $u + d' \in I_i(n_i)$, for all $i = 1, \ldots, k$ and for all $d' \leq d$, $d' \in \mathbb{R}_0^+$.
- $(\overline{n}, u) \xrightarrow{a} (\overline{n'}, u')$ (internal action rule) if and only if there is an edge $n_i \xrightarrow{g,a,r}_{s} n_i'$, for some $i \in \{1, \ldots, k\}$, such that $n_j' = n_j$, for all $j \neq i$, $u \in g$, $u' = (u\{s\})|_r$, and $u' \in \bigwedge_{h=1,\ldots,k} I_h(n_h')$.
- $(\overline{n}, u) \xrightarrow{m} (\overline{n'}, u')$ (synchronization rule) if and only if there exist i, j, $i \neq j$, such that:

(1) $n_h' = n_h$, for all $h \neq i, h \neq j$.
(2) There exist two edges $n_i \xrightarrow[s_i]{g_i, m!, r_i} n_i'$ and $n_j \xrightarrow[s_j]{g_j, m?, r_j} n_j'$, such that $u \in g_i \wedge g_j$, $u' = (((u\{s_i\})\{s_j\})|_{r_i})|_{r_j}$.
(3) $u' \in \bigwedge_{h=1,\ldots,k} I_h(n_h')$. □

From this definition, we can easily define the *timed traces* of an NTA as the sequences of both delays and actions $t \in (\mathbb{R}_0^+ \cup \Sigma)^*$ 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 [4] 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.

5 Translating WS-CDL documents into Timed Automata

A function $\varphi : \mathit{Activities} \times \mathcal{P}_{\mathcal{F}}(C) \times \mathcal{N} \longrightarrow \mathcal{NTA} \times \mathcal{P}_{\mathcal{F}}(C)$ 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 a set of 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.

[4] Including the empty trace ϵ.

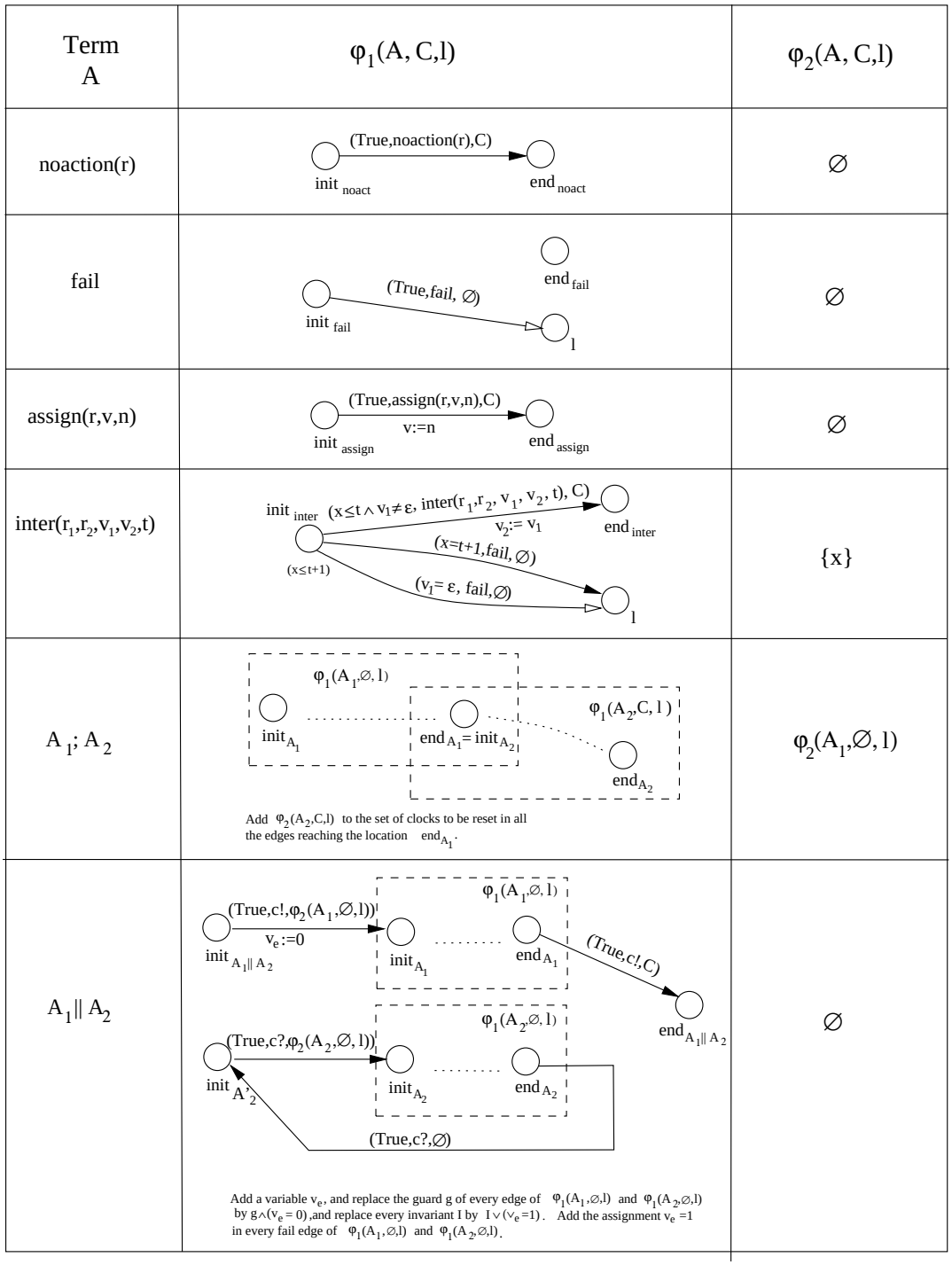

Fig. 1. From WS-CDL to NTA (I)

We will denote by $\varphi_1(A, C, l)$ the first projection of φ, 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 (Figure 3):

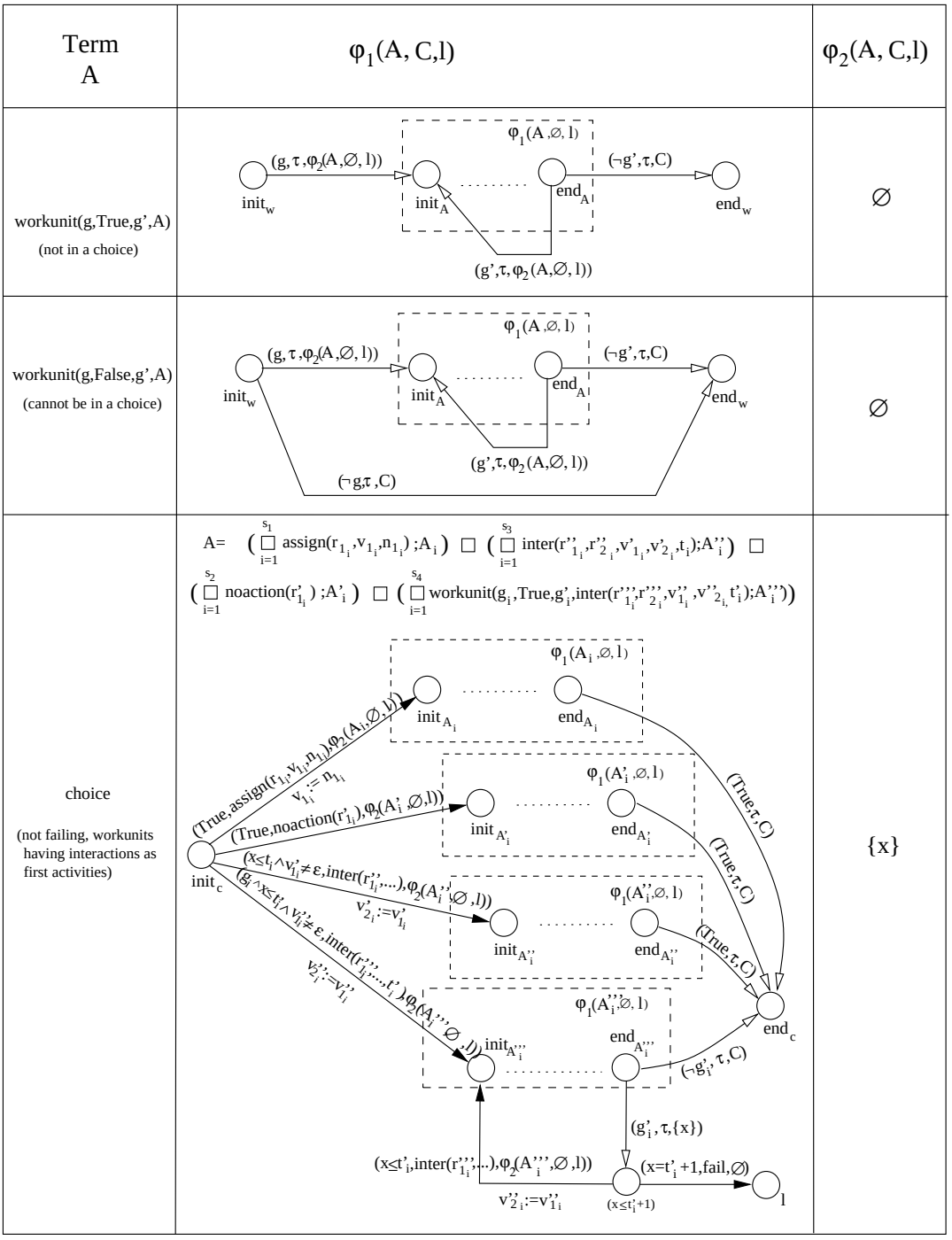

Fig. 2. From WS-CDL to NTA (II)

- 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 $\varphi(A_2, \emptyset, de)$.
- 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 $\varphi(A_1, \emptyset, e)$.

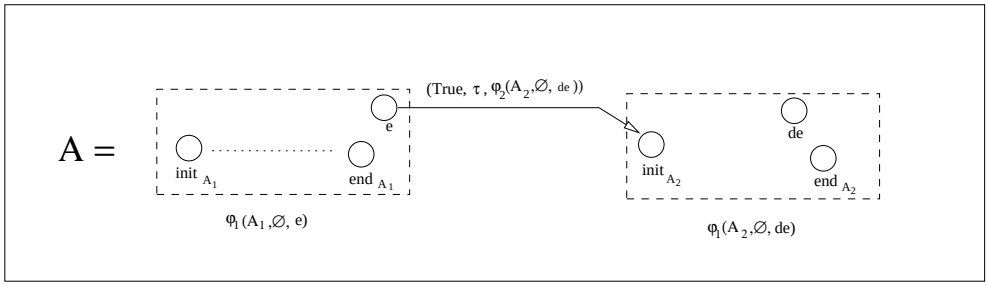

Fig. 3. From WS-CDL to NTA (III)

- We connect the exception location 'e' with the initial location of $\varphi_1(A_2, \emptyset, de)$ by means of an urgent edge, which must reset all the clocks in $\varphi_2(A_2, \emptyset, de)$.

Figures 1 and 2 show how the function φ 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 φ, and reset the clocks indicated as third parameter in all the edges reaching the final location.

We omit a formal definition of the NTA produced as result of the application of φ, 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**$(\mathbf{r_1}, \mathbf{r_2}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{t})$: 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.
- $\mathbf{A_1}\,;\mathbf{A_2}$: we first obtain the corresponding NTA both for A_1 and A_2, as indicated in Figure 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 $\varphi_1(A_2, C, l)$. Notice that all the edges reaching this node must reset all the clocks in $\varphi_2(A_2, C, l)$, and also that the set of clocks to be reset when using the generated NTA is that of A_1.
- $\mathbf{A_1} \parallel \mathbf{A_2}$: we first obtain the corresponding NTA both for A_1 and A_2,

as indicated in Figure 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 I are replaced by $(I \vee v_e = 1)$, 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, 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 τ, which resets the clocks in 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 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, specially 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. The WS-CDL document description in [17] has nothing to say about this specific case, probably because it would be rare to find a parallel activity as an alternative in a choice. Actually, the translation for this case would require a rather large distinction of subcases. Thus, taking into account that this composite construction will be very unusual, we have decided to ban it in the translation.

  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.

  Figure 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,

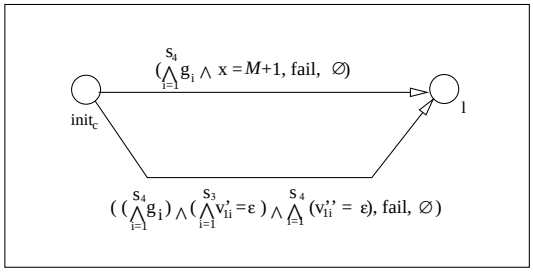

Fig. 4. Fail Edges in a Choice

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 $M = Max(t_1, \ldots, t_{s_3}, t'_1, \ldots, t'_{s_4})$, or no source variable of these interactions has a value assigned. In Figure 4 we depict the two urgent edges that we should add in this case [5].

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 *fail*□*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 *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 *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 $[\overline{A}]_{\equiv}$ there is a *Var-contextual timed trace* s' of $t(A)$ such that $\phi(s) = \phi'(s')$, where ϕ is a function that removes from s all the internal movements (empty transitions), and ϕ' a function that removes from s' both the τ-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 $[\overline{A}]_{\equiv}$ such that $\phi(s) = \phi'(s')$.

Proof: We use structural induction on A:

- **Base cases**: These are the *assign*, *noaction*, *inter* and *fail*. For all of them

[5] If $M = \infty$ the time-out edge would not appear.

the result is immediate, just observing the contextual timed traces that we can obtain with both semantics. For instance, in the case of *inter*, according to the operational semantics we may have a failure in time zero when $v_1 = \epsilon$, which is also the case in the automaton, due to the urgent edge labelled with *fail*, and also, for $t < \infty$ we have a time-out failure in time $t+1$, which is also captured by the automaton. Finally, the normal execution of the interaction within the time t can be performed in both cases.

- **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 ϕ, ϕ'. Let us then see the different cases we have:
  - *Sequence*: According to the operational semantics, the contextual timed traces of $[A_1; A_2]_{\equiv}$ are obtained as the concatenation of their contextual timed traces, except in the event of a failure by $[A_1]_{\equiv}$, in which case the timed trace terminates immediately with the *fail* action, and then, the only possible evolution is time elapsing. The same occurs in the generated timed automata, since we are collapsing in a single location the final location of $t(A_1)$ and the initial one of $t(A_2)$, and also, all the clocks that $t(A_2)$ needs to be reset are actually reset by the edges reaching that location. Furthermore, in the event of a failure the control flow is immediately transferred to the location l (by induction hypothesis).
  - $A_1 \| A_2$: 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. The same occurs with the *Var*-contextual timed traces in the corresponding NTA (Figure 1), although there are both one initial and one final synchronization over the channel c (which are hidden by ϕ'). Notice that we are not only considering the timed traces that each activity can generate isolately, 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. This is not the case for the new variable v_e, which does not affect the behaviour of both automata as long as no *fail* transition was executed. This is because the additional condition $v_e = 0$ introduced in all the edge guards would always be satisfied, and also because the OR-condition introduced in the locations with some invariant $(I \vee v_e = 1)$ does not perturb the previous invariant condition I, except when a fail transition is executed, in which case v_e changes its value to 1, wich prevents any enabled transitions in the other automaton from being executed. Time elapsing is, however, still possible in the current location of this automaton, due to the change introduced in the invariants.
  - *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. In the operational semantics me have

a null transition that activates the execution of the actions in A, which is hidden by ϕ, and in the NTA we have an urgent edge labelled with τ, which is also hidden by ϕ'. Something similar occurs for the evaluation of the repetition condition, g', although in this case when it is false we immediately abandon the workunit construction, using a null transition in the operational semantics and an urgent τ-edge in the NTA. When *block* is false, time cannot elapse in both cases, so if g is false the workunit is immediately abandoned (again, by a null transition in the operational semantics and an urgent τ-edge in the NTA).

· *Choice*: we use an extended choice with the syntax indicated in Figure 2. From the operational semantics it follows that any alternative can proceed by executing its first activity, and this is also captured by the NTA depicted in that Figure. Notice that a choice can only execute a fail transition when all its arguments are able to do so, because when an *assign* or a *noaction* activity appears as alternative, there cannot be any possible fail transition in the choice. Otherwise, the fail transition in the choice could only be produced either by the time-out of all the alternative interactions, or because no source variable is available in these interactions. Both cases are covered by the edges indicated in Figure 4.

□

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 Case Study: List of Registered Voters Management

The use of Web Services for e-government has become more and more important in the recent 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 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 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

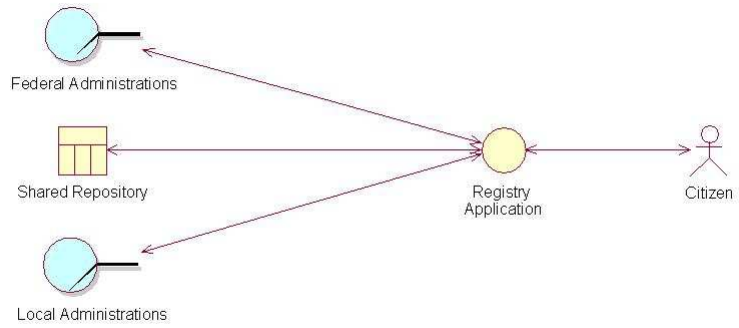

Fig. 5. Diagram of the system

elections.

We focus on the case of a citizen who decides to inscribe himself in these lists. In Figure 5 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 inscribe himself in the lists of registered 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 minutes 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. The information about the citizen and the procedures he could used is 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.

7.1 WSCDL Generation Phase

In Figure 6 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.

Figure 7 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 two interactions executed in parallel

```
...
<sequence>
    <interation name="Login of the citizen in the system"
     channelVariable="Registry2RepositoryChannel"
     operation="LoginInSystem">
     <description type="description">
      Sending the citizen user and password to Repositoy
     </description>
     <participate relationshipType= "RegistryRepository"
      fromRole="Registry Application"
      toRole="Shared Repository"/>
     <exchange name= "LoginUser"
      action="request">
      <use variable="Login"/>
      <populate variable="User"/>
     </exchange>
    </interaction>
    <parallel>
    <interation name="Information about the citizen from Repository"
     channelVariable="Registry2RepositoryChannel"
     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>
    <interation name="Information about procedures from Repository"
     channelVariable="Registry2RepositoryChannel"
     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. 6. WS-CDL specification for the case study (I)

after A, but before C. Letters E_1 and E_2 correspond to the options for the different kinds of citizens that can be executing the process. Finally, letter F refers to the interaction with a local administration, while letter G refers to the interaction with a federal administration.

Figure 8 shows the translation into timed automata by applying the rules described in Section 5. We can distinguish two automata in this Figure: the **Main** automaton corresponding to the whole choreography and the **Parallel** automaton that implements the parallel interaction B_2.

$$
\begin{aligned}
&\mathit{RegisteredVoters} = A;(B_1 \| B_2);C;D;(E_1 \Box E_2)\\
&A = \quad \mathit{inter}(\mathit{Registry}, \mathit{Repository}, \mathit{Login}, \mathit{User}, \infty)\\
&B_1 = \quad \mathit{inter}(\mathit{Repository}, \mathit{Registry}, \mathit{UserData}, \mathit{CitizenInfo}, \infty)\\
&B_2 = \quad \mathit{inter}(\mathit{Repository}, \mathit{Registry}, \mathit{ProceduresData}, \mathit{ProceduresInfo}, \infty)\\
&C = \quad \mathit{inter}(\mathit{Registry}, \mathit{Repository}, \mathit{SelOption}, \mathit{Option}, 5)\\
&D = \quad \mathit{inter}(\mathit{Repository}, \mathit{Registry}, \mathit{List}, \mathit{ListInfo}, \infty)\\
&E_1 = \quad \mathit{workunit}(\mathit{CitizenType} == \mathit{EuropeanInSpain}, \mathit{true}, \mathit{false}, F)\\
&E_2 = \quad \mathit{workunit}(\mathit{CitizenType} == \mathit{SpanishOutSpain}, \mathit{true}, \mathit{false}, G)\\
&F = \quad \mathit{inter}(\mathit{Registry}, \mathit{Local}, \mathit{CitizenInfo}, \mathit{Citizen}, \infty)\\
&G = \quad \mathit{inter}(\mathit{Registry}, \mathit{Federal}, \mathit{CitizenInfo}, \mathit{Citizen}, \infty)
\end{aligned}
$$

Fig. 7. Algebraic Specification of the case study

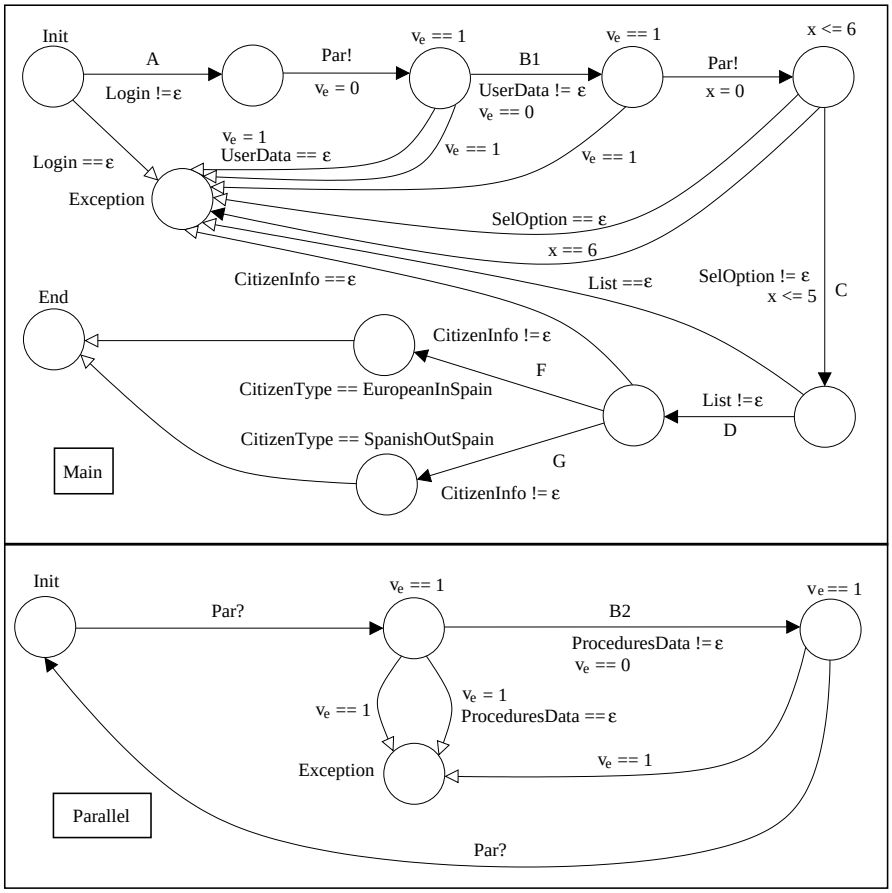

Fig. 8. Timed automata for the case study

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 minutes doing nothing after login correctly. The query is specified in the following way in UPPAAL:

$$(Main.Init\_Session \wedge x > 5) - - > Main.Exception$$

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:

$$A[] \quad Main.After\_F \quad imply \quad Main.CitizenType == 1$$

We obtain that this formula is **satisfied**.

(3) *Spanish in Session S.* This property is complementary to the property before. 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:

$$A[] \quad Main.After\_G \quad imply \quad Main.CitizenType == 2$$

We obtain that this formula is **satisfied**.

(4) *Ends On Time.* Finally, we want to prove that the process finishes in 6 minutes at most, i.e., interaction F or interaction G is executed in 6 minutes after login. Otherwise, an exception will be thrown up. We call **Before_Choice** the interaction before executing interaction F or interaction G, so the query in UPPAAL for this property is:

$$(Main.Before\_Choice \wedge x > 6) - - > Main.Exception$$

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 fulfill the last property. The solution is adding a time-out of 6 minutes to both, interaction F ("Send Inscription to Local Admin") and interaction G ("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 minutes without throwing up an exception.

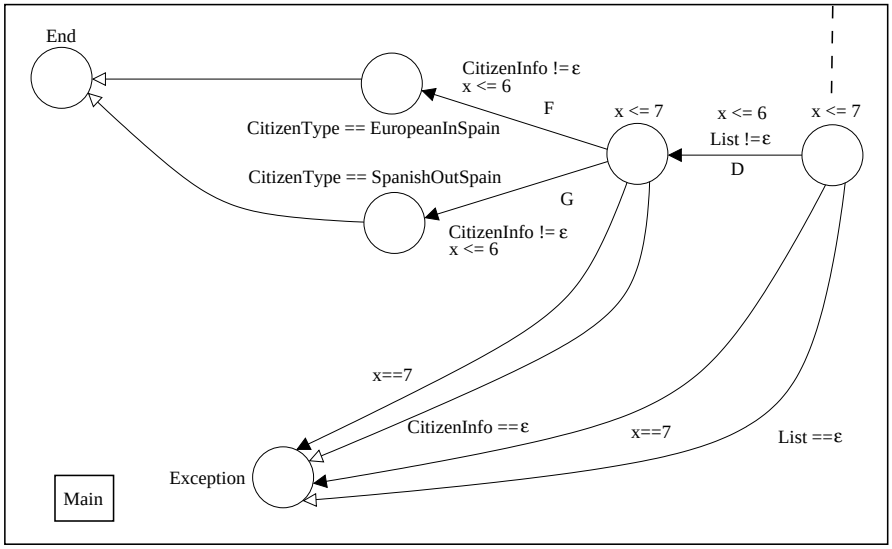

Fig. 9. Main timed automaton modified

Figure 9 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 four properties described before with the UPPAAL tool. Now, we obtain that the automata do **satisfy** all the requirements.

8 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 parties 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 very similar syntax to 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, then, is the formalization of WS-CDL semantics, which is presented in a textual way in [17], 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.

Finally, 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.

References

[1] A. Ankolenkar et al. DAML-S: Web Service Description for the Semantic Web. *Proc. First International Semantic Web Conference*, pages 348–363, 2002.

[2] A. Arkin et al. Web Service Choreography Interface (WSCI) 1.0. In http:/www.w3.org/TR/wsci/.

[3] A. Arkin et al. Web Services Business Process Execution Language Version 2.0, Dec. 2004. http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm.

[4] R. Alur and D. Dill. Automata For Modeling Real-Time Systems. In *ICALP*, pages 322–335, 1990.

[5] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing Web Service Choreographies. *Electr. Notes Theor. Comput. Sci.*, 105:73–94, 2004.

[7] M. Carbone, K. Honda, and N. Yoshida. A Theoretical Basis of Communication-Centres Concurrent Programming. In *First International Summer School on Emerging Trends in Concurrency (TIC'06)*, 2006.

[8] G. Díaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Verification of Web Services with Timed Automata. In *Proceedings of First International Workshop on Automated Specification and Verification of Web Sites, Electronic Notes in Theoretical Computer Science, vol. 1125, 157/2*, 2005.

[9] J. S. Dong, Y. Liu, J. S. 0001, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 226–245, 2006.

[10] X. Du, C. Xing, L. Zhou, and Z. Li. Nested Web Service Interface Control Flow Automata. *In Proc. 4th International Conference on Next Generation Web Services Practices*, pages 129–136, 2008.

[11] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model Checking Service Compositions Under Resource Constraints. In *ESEC/SIGSOFT FSE*, pages 225–234, 2007.

[12] R. Hamadi and B. Benatallah. A Petri Net-based Model for Web Service Composition. *Proc. 14th Australasian Database Conference*, 17:191–200, 2003.

[13] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[14] N. Lohmann and J. Kleine. Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes. In *Modellierung*, pages 57–72, 2008.

[15] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[16] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, part I and II. *Information and Computation*, 100(1):1–40, 1992.

[17] N. Kavantzas et al. Web Service Choreography Description Language (WSCDL) 1.0. http://www.w3.org/TR/ws-cdl-10/.

[18] S. Narayanan and S. Mcllraith. Simulation, Verification and Automated Composition of Web Services. *Proc. 11th International World Wide Conference*, pages 77–88, 2002.

[19] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. 2nd International Conference on Web Services (ICWS)*, pages 43–, 2004.

[20] N. Sharygina and D. Kröning. Model Checking with Abstraction for Web Services. In *Test and Analysis of Web Services*, pages 121–145. Springer, 2007.

[21] V. Valero, M. Cambronero, G. Díaz, and H. Macià. A Petri Net Approach for the Design and Analysis of Web Services Choreographies. *Journal of Logic and Algebraic Programming*, To appear, 2009.

[22] H. Yang, X. Zhao, C. Cai, and Z. Qiu. Model-Checking of Web Services Choreography. *IEEE International Workshop on Service-Oriented System Engineering*, pages 79–84, 2008.

[23] W. L. Yeung. Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services. In *European Conference on Web Services (ECOWS)*, pages 297–305, 2006.