Analysis and Verification of Time Requirements applied to the Web Services Composition [⋆]

Gregorio Díaz, María-Emilia Cambronero, M Llanos Tobarra
Valentín Valero and Fernando Cuartero

Department of Computer Science
University of Castilla-La Mancha
Escuela Politécnica Superior de Albacete. 02071 - SPAIN
[gregorio,emicp,llanos,valentin,fernando]@dsi.uclm.es

Abstract. This work presents a new approach to the analysis and verification of the time requirements of Web Services compositions via goal-driven models and model checking techniques. The goal-driven model used is an extension of the goal model KAOS and the model checker engine is the UPPAAL tool. The goal model specifies the properties that the system must satisfy and how they should be verified by using the model checker engine. In order to illustrate this approach, we apply these techniques to a basic Internet purchase process.

1 Introduction

A basic activity in the design of software system and by extension to Web Services is the analysis and verification of the requirements that the system must satisfy. However, before performing the analysis and verification, the software engineer must gather these requirements in an standardized specification.

In this work, we have focused our efforts on those systems where time plays an important role. Thus, in the literature we can find related works for the specification of software system requirements, as for instance [17]. However, we have based this work on the work of Lamsweerde et al [1,8,20]. Thus we have extended this work in order to describe more complex goal-driven requirement models. Once we have captured the system requirements and implemented the system by means of Web Services composition [5,4] (concretely by using the Web Service Choreography Description Language, WS-CDL [12]), these Web Services are translated into Timed Automata [2] by using the technique presented in [11]. After the translation, we can verify the time requirements by using the model checker, UPPAAL [9,10,15].

This work is structured in seven sections. In the first section we have already seen a brief introduction to the work. The second section presents the methodology approach. The third section specifies the study case that we follow in this work. The fourth section shows a goal-driven model for gathering

[⋆] This work has been supported by the CICYT project “Description and Evaluation of Distributed Systems and Application to Multimedia Systems”,TIC2003-07848-C02-02 and the UCLM project “Aplicación de Métodos Formales al Desarrollo y Verificación de Web Services”

time requirements. The fifth section performs a brief summary about WS-CDL, timed automata and the translation process between them. In the sixth section, we will see how to perform the verification process. Finally, the seventh section deals with the conclusions and future works.

2 The Methodology Approach

The proposed methodology is divided into three phases (Fig. 1): Analysis, design and verification. The analysis is performed by using an extension of the goal model KAOS. This goal model allows analysts and specifiers to gather time requirements of software systems in a hierarchic order, i.e., from general and strategic goals to concrete requirements.

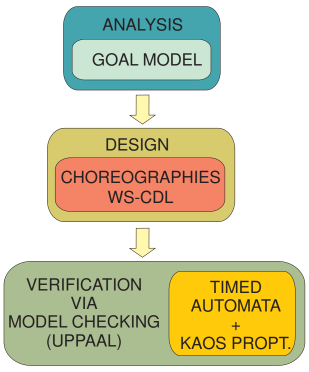


Fig. 1. The proposed Methodology for Web Services composition.

The design is performed via composed specifications written in Web Services choreographies (WS-CDL), which are known as Web Services choreography specifications. These specifications appear as a necessary result of composing Web Services and implement mechanisms to deal with compositional problems, as for instance concurrency and time aspects.

The verification phase in the literature is useful taken together with the design phase. However, during the last few years, there has been a growing consensus that verification is a key instrument for developing software systems, in that sense Hoare [14], Clarke [7] together with a large number of authors have agreed in its importance [13]. Thus, we have considered that the verification is substantial enough to be taken apart from previous phases, although, we should not forget how close this phase is to the design. In this phase we have used a translation algorithm presented in [11] in order to translate the choreographies specified in the design into timed automata, which are the formalism used by the model checker UPPAAL. The timed automata captures in a proper manner the time behaviors of the different Web Services involved in a choreography. Once this translation is successfully finished, the verifiers can check whether the requirements, now transformed into properties, are fulfilled by the timed automata or not. If the verifiers find that the timed automata do not satisfy a property, then they can use the counterexample obtained from the verification to locate where exactly the error lies. This error can occur for several reasons, in which are included: Requirement specification errors, choreography specification errors and, the most desirable case, errors in the real system.

3 The study case: An Internet Purchase Process

This example is based upon a typical purchase process that uses Internet as a business context for a transaction. There are three actors in this example: a customer, a seller and a carrier. The Internet purchase works as follows: *"A customer wants to buy a product by using Internet. There are several sellers that offer different products in Internet Servers based on Web-pages. The customer contacts a seller in order to buy the desired product. The seller checks the stock and contacts with a carrier. Finally, the carrier delivers the product to the customer."*

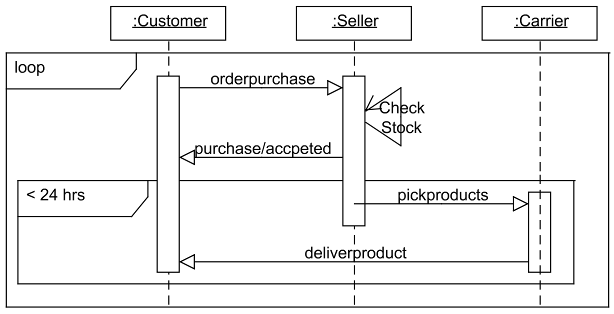


Fig. 2. The diagram for a purchase process by Internet

Figure 2 depicts the diagram that represents this purchase process. This process consists of three participants: the customer, the seller and the carrier. The behavior of each participant is defined as follows:

- Customer: He contacts the seller to buy a product. He must send the seller the information about the product and the payment method. After the payment, he waits to receive the product from a carrier within the agreed time, twenty four hours.
- Seller: He receives the customer order and the payment method. The seller checks if there is enough stock to deliver the order and sends an acceptance notification to the customer . If there is stock to deliver the order, then he contacts with a carrier to deliver the product.
- Carrier: He picks up the order and the customer information in order to deliver the product to the customer. The interval to deliver the product is the time that the seller has stipulated, one day, which is the main temporal constraint.

4 The analysis phase

The requirements, properties and characteristics of the system must be gathered in order to be checked. However, they must be expressed in a formalized manner. There are several languages, graphical diagrams, etc. to perform this, but we apply those in which time requirements are well captured. In this sense, goal-oriented requirements engineering emerges as a natural choice. The key activity in goal-oriented requirements engineering is the construction of the goal model. Goals are objectives the system under construction must achieve. Goal formulations thus refer to intended properties to be ensured. They are formulated at different levels of abstraction from high-level, strategic concerns to low-level technical concerns. Goal models also allow analysts to capture and explore alternative refinements for a given goal. The resulting structure of the goal model is an AND-OR graph. The specific goal-oriented framework considered here is an extension of KAOS methodology [1,6,8,20] which has a two-level language: (1) an outer semi-formal layer for capturing, structuring and presenting requirements engineering concepts; (2) an inner formal assertion layer for their precise definition and for reasoning about them.

4.1 The inner formal assertions layer: TCTL style requirements

The formal assertions, in which the goals are written, use the UPPAAL language for specifying properties. This language is a subset of timed computation tree logic (TCTL) [19,18], where atomic expressions are location names, variables and clocks from the modeled system. The properties are defined using local properties that are either true or false depending on a specific configuration.

Definition 1 (Local Property) *Given an UPPAAL model* $\langle \boldsymbol{A}, Vars, Clocks, Chan, Type \rangle$*. A formula* φ *is a local property iff it is formed according to the following syntactical rules:*

$$\begin{array}{ll}
\varphi ::= deadlock & \\
\quad \mid A.l & for\ A \in \boldsymbol{A}\ and\ l \in L_A \\
\quad \mid x \bowtie c & for\ x \in Clocks, \bowtie \in \{<,<=,==,>=>\}, c \in \mathbb{Z} \\
\quad \mid x - y \bowtie c & for\ x, y \in Clocks, \bowtie \in \{<,<=,==,>=>\}, c \in \mathbb{Z} \\
\quad \mid a \bowtie b & for\ a, b \in Vars \bigcup \mathbb{Z}, \bowtie \in \{<,<=,!=,==,>=>\} \\
\quad \mid (\varphi_1) & for\ \varphi_1\ a\ local\ property \\
\quad \mid\ not\ \varphi_1 & for\ \varphi_1\ a\ local\ property \\
\quad \mid \varphi_1\ or\ \varphi_2 & for\ \varphi_1, \varphi_2\ logical\ properties\ (logical\ OR) \\
\quad \mid \varphi_1\ and\ \varphi_2 & for\ \varphi_1, \varphi_2\ logical\ properties\ (logical\ AND) \\
\quad \mid \varphi_1\ imply\ \varphi_2 & for\ \varphi_1, \varphi_2\ logical\ properties\ (logical\ implication)
\end{array}$$

In Definition 1 we have expressed the syntaxis of the temporal logic that UPPAAL uses. Now, let us see the definition of the five different property classes that UPPAAL may check.

Definition 2 (Temporal Properties) *let* $M = \langle \boldsymbol{A}, Vars, Clocks, Chan, Type \rangle$ *be an UPPAAL model and let* φ *and* ψ *be local properties. The correctness of temporal properties is defined for the classes* $A[\,]$*,* $A <>$ *and* $-->$ *as follows:*

$$\begin{array}{ll}
M \vDash A[\,]\ \varphi & iff\ \forall \{(\boldsymbol{l}, e, v)\}^K \in \tau(M).\ \forall k \le K.\ (\boldsymbol{l}, e, v)^k \vDash_{loc} \varphi \\
M \vDash A <>\ \varphi & iff\ \forall \{(\boldsymbol{l}, e, v)\}^K \in \tau(M).\ \exists k \le K.\ (\boldsymbol{l}, e, v)^k \vDash_{loc} \varphi \\
M \vDash \varphi --> \psi & iff\ \forall \{(\boldsymbol{l}, e, v)\}^K \in \tau(M).\ \forall k \le K \\
 & \quad (\boldsymbol{l}, e, v)^k \vDash_{loc} \varphi \Rightarrow \exists k' \ge k.\ (\boldsymbol{l}, e, v)^{k'} \vDash_{loc} \psi
\end{array}$$

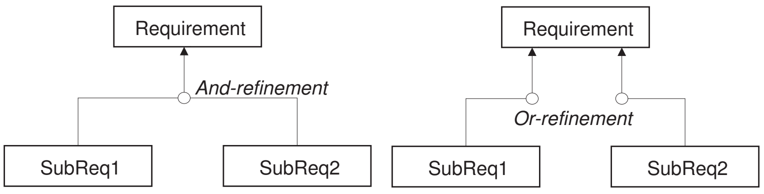


Fig. 3. And-refinement and Or-refinement goal models.

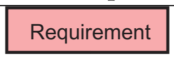
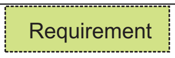
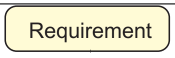
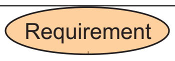
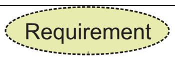
The two temporal property classes dual to A[ ] and A <> are defined as follows:

$$M \vDash E <> \varphi \text{ \textit{iff} } \neg(M \vDash_{loc} A[\,]\ not(\varphi))$$
$$M \vDash E[\,]\ \varphi \quad \text{\textit{iff} } \neg(M \vDash_{loc} A <> not(\varphi))$$

4.2 The outer semi-formal layer: the goal-driven model

Two key elements are used as building elements for the definition of a goal model: *goals* and *requirements*. A goal prescribes intended behaviors of the system. It may refer to services to be provided (functional goals) or to the quality of service (non-functional goals). A requirement is a leaf goal that requires cooperation between different parties, which are called *agents*. Agents are active components that play a role in achieving goal satisfaction. To build Goal Models, goals are organized in an AND/OR refinement - abstraction hierarchy where higher-level goals are, in general, strategic, coarse-grained and involve multiple agents whereas lower-level goals are, in general, technical, fine-grained and involve fewer agents. In such structures, AND-refinement links relate a goal to a set of subgoals (called refinement) possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal, as seen in the left-hand side of Figure 3. OR-refinement links may relate a goal to a set of alternative refinements, as seen in right-hand side of Figure 3.

Requirements must be checked by the model checker and are formalized in a real-time temporal logic that we have shown above. Keywords such as *Achieve* (reachability), *Avoid* (not safety), *Maintain* (safety), *possibly always*, *inevitably* and *unbounded response*, are used to name goals according to the temporal behavior pattern they prescribe. They are depicted in the goal model as follows:

Temporal Behavior	**Goal Model Representation**
Maintain (Safety) $A[\,]\ \varphi$	
Achieve (Reachability) $E <> \varphi$	
Possibly Always $E[\,]\ \varphi$	
Inevitably $A <> \varphi$	
Unbounded Response $\varphi - - > \psi$	

Once we have defined the goal model, we can apply this technique to our example. We must identify the crucial requirements for the Internet purchase process that we have described above. For this we have identified two different kinds of requirements. One kind refers to the obligation that both the seller and carrier have agreed to deliver the product on time, while the other refers to the quality of service. The time restriction establishes that the seller and carrier have twenty four hours to deliver the product. So, the seller must prepare the order for the carrier to send the product within the interval. The service quality is determined by two different requirements that are closely linked. The service must be rapid and also efficient. Due to this close relationship between these two requirements, if one of them is fulfilled then the other is fulfilled too.

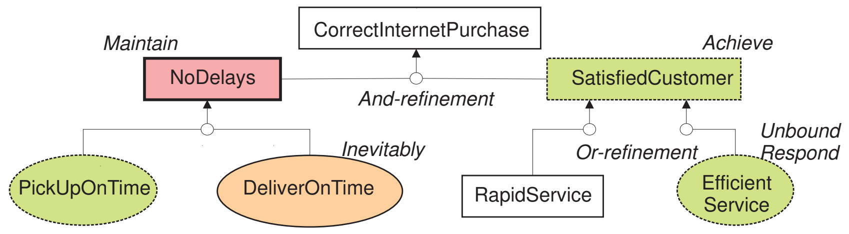


Fig. 4. The goal-model for the Internet Purchase Process

Figure 4 depicts the goal-model that we have developed for this example. The root goal *"CorrectInternetPurchase"* is decomposed into two subgoals by an And-refinement, which means that each one must be fulfilled in order to achieve the root goal. The first one, *"NoDelays"*, that is of type "maintain", is refined by another And-refinement with two leaf goals that inherit the maintain character. The first leaf goal *"PickupOnTime"* is of type "Unbound Respond". This goal represents the situation that the carrier must pick up the order on time and is formalized as follows:

$$\begin{gathered} Customer.WaitOrder - - > \\ (Carrier.PickUp \wedge Clock_{deliver} < 24hours) \end{gathered} \tag{1}$$

The second leaf goal *"DeliverOnTime"* is of type "Inevitably" and specifies that the carrier must deliver the order on time. The goal is defined as follows:

$$A <> (Carrier.Deliver \wedge Clock_{deliver} < 24hours) \tag{2}$$

The second one, *"SatisfiedCustomer"*, of type "Achieve", is formed by two leaf goals. These leaf goals refine the parent goal by an Or-refinement, which means that if one of them is satisfied then the parent goal is satisfied too. The leaf goal *"RapidService"*, that determines that the customer will receive the order on time, is specified as follows:

$$E <> (Customer.ReceiveOrder \wedge Clock_{deliver} < 24hours) \tag{3}$$

The leaf goal *"EfficientService"* has the behavior of an "Unbounded Response" requirement. This goal indicates that when the seller accepts the order, then in the future, the customer will receive the order. This goal is formalized as follows:

$$Seller.AcceptOrder - - > Customer.ReceiveOrder \tag{4}$$

5 The design phase

In the design phase, designers must specify the system by implementing it with the Web Service Choreography Description Language. Once we have this choreography specification, we can use the work presented in [11] in order to obtain the equivalent timed automata.

5.1 Designing Web Services composition with WS-CDL

WS-CDL describes interoperable collaborations between parties. In order to facilitate these collaborations, services commit to mutual responsibilities by establishing Relationships. Their collaboration takes place in a jointly agreed set of ordering and constraint rules, whereby information is exchanged between the parties. The WS-CDL model consists of the following entities:

- **Participant Types, Role Types and Relationship Types** within a Choreography. Information is always exchanged between parties within or across trust boundaries. A Role Type enumerates the observable behavior a party exhibits in order to collaborate with other parties. A Relationship Type identifies the mutual commitments that must be made between two parties for them to collaborate successfully. A Participant Type groups together those parts of the observable behavior that must be implemented by the same logical entity or organization.
- **Information Types, Variables and Tokens.** Variables contain information about commonly observable objects in a collaboration, such as the information exchanged or the observable information of the Roles involved. Tokens are aliases that can be used to reference parts of a Variable. Both Variables and Tokens have Types that define the structure of what the Variable contains or the Token references.

- **Choreographies** define collaborations between interacting parties:
  - **Choreography Life-line**: This shows the progression of a collaboration. Initially, the collaboration is established between the parties; then, some work is performed within it, and finally it completes either normally or abnormally.
  - **Choreography Exception Block**: This specifies the additional interactions that should occur when a Choreography behaves in an abnormal way.
  - **Choreography Finalizer Block**: This describes how to specify additional interactions that should occur to modify the effect of an earlier successfully completed Choreography (for instance to confirm or undo the effect).
- **Channels** establish a point of collaboration between parties by specifying where and how information is exchanged.
- **Work Units** prescribe the constraints that must be fulfilled for making progress and thus performing actual work within a Choreography.
- **Activities and Ordering Structures.** Activities are the lowest level components of the Choreography that perform the actual work. Ordering Structures combine activities with other Ordering Structures in a nested structure to express the ordering conditions in which information within the Choreography is exchanged.
- **Interaction Activity** is the basic building block of a Choreography, which results in an exchange of information between parties and possible synchronizations of their observable information changes, and the actual values of the exchanged information.

Figure 5 shows a piece of the WS-CDL specification corresponding to this purchase process.

5.2 Timed Automata

By definition, a timed automaton is a standard finite-state automaton extended with a finite collection of real valued clocks. The clocks are assumed to proceed at the same rate and their values may be compared with natural numbers or reset to 0. UPPAAL extends the notion of timed automata to include integer variables, i.e. integer valued variables that may appear freely in general arithmetic expression used in guards as well as in assignments.

The model also allows clocks not only to be reset, but also to be set to any non-negative integer value.

Definition 3 (Atomic Constraints) *Let C be a set of real valued clocks and I a set of integer valued variables. An atomic clock constraint over C is a constraint of the form: $x \sim n$ or $x - y \sim n$, for $x, y \in C$, $\sim \in \{\leq, \geq, =\}$and $n \in \mathbf{N}$. An atomic constraint over I is a constraint of the form: $i \sim n$, for $i \in I$, $\sim \in \{\leq, \geq, =\}$and $n \in \mathbf{Z}$.*

```
<interaction name="createPO" channelVariable="tns:seller-channel"
             operation="handlePurchaseOrder" align="true" initiate="true">
  <participate relationshipType="tns:CostIntSellCarrRS"
               fromRole="tns:Customer" toRole="tns:Seller"/>
  <exchange name="request" informationType="tns:purchaseOrderType"
            action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
  <exchange name="response" informationType="purchaseOrderAccepted"
            action="respond">
    <send variable="cdl:getVariable("tns:purchaseOrderAcceted","","")"/>
    <receive variable="cdl:getVariable("tns:purchaseOrderAccepted","","")"/>
  </exchange>
  <exchange name="NoStockAckException" informationType="NoStockAckType"
            action="respond">
    <send variable="cdl:getVariable('tns:NoStockAck', '', '')"
          causeException="true" />
    <receive variable="cdl:getVariable("tns:NoStockAck","","")"
             causeException="true"/>
  </exchange>
  <record name="record-the-channel-info" when="after">
    <source variable="cdl:
                  getVariable("tns:purchaseOrder,"","PO/CustomerRef")"/>
    <target variable="cdl:getVariable("tns:customer-channel", "", "")"/>
  </record>
  <record name="reset-clock" when="after">
    <source variable="00:00"/>
    <target variable="cdl:getVariable("tns:Clock1", "", "")"/>
  </record>
</interaction>
<interaction name="PickUpProductPO" channelVariable="tns:deliver-channel"
             operation="PickUpPurchaseOrder" align="true" initiate="true">
  <participate relationshipType="tns:CustIntSellCarrRS"
               fromRole="tns:Seller" toRole="tns:Carrier"/>
  <exchange name="request"
            informationType="tns:purchaseOrderType" action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
</interaction>
<interaction name="DeliverProductPO" channelVariable="tns:customer-channel"
             operation="DeliverProductOrder" align="true" initiate="true">
  <participate relationshipType="tns:CostIntSellCarrRS"
               fromRole="tns:Carrier" toRole="tns:Customer"/>
  <exchange name="request" informationType="tns:purchaseOrderType"
            action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
  <timeout  time-to-complete=
             "cdl:minor(cdl:getVariable("tns:Clock1","",""),"48:00")"/>?
</interaction>
```

Fig. 5. WS-CDL interaction specification of the Internet purchase process

By $C_c(C)$ we will denote the set of all clock constraints over C, and by $C_i(I)$ we will denote the set of all integer constraints over I.

Definition 4 (Guards) *Let C be a set of real valued clocks, and I a set of integer valued variables. A guard g over C and I is a formula generated by the following syntax: $g ::= c|g \wedge g$, where $c \in (C_c(C) \bigcup C_i(I))$.*

$\mathcal{B}(C, I)$ will stand for the set of all guards over C and I.

Definition 5 (Assignments) *Let C be a set of real valued clocks and I a set of integer valued variables. A clock assignment over C is a tuple $\langle v, c\rangle$, where $v \in C$ and $c \in \mathbf{N}$. An integer assignment over I is a tuple $\langle w, d\rangle$ representing the assignment $w = d$, where $w \in I$ and $d \in \mathbf{Z}$.*

We will use $\mathcal{A}(C, I)$ to denote the power-set of all assignments over I and C.

Definition 6 (Timed automata) *A timed automaton A over a finite set of actions Act, clocks$C$ and integer variables I is a tuple $\langle L, l_0, E, V\rangle$, where L is a finite set of nodes (control-nodes), l_0 is the initial node, $E \subseteq L \times \mathcal{B}(C, I) \times Act \times \mathcal{A}(C, I) \times L$ corresponds to the set of edges, and $V : L \rightarrow \mathcal{B}(C, I)$ assigns invariants to locations. For a brief notation, we will denote $l \xrightarrow{g,a,r} l'$ by the edge $\langle l, g, a, r, l'\rangle \in E$.*

5.3 Translation process: WS-CDL into Time Automata

For each component of a WS-CDL description we have the following correspondence in timed automata (see Fig. 6 for a schematic presentation of this correspondence):

Role : These are used to describe the behavior of each class of party that we are using in the choreography. Thus, this definition matches with the definition of a *template* in timed automata terminology.

Relation type : These are used to define the communications between two roles, and the needed channels for these communications. In timed automata we just need to assign a new channel for each one of these channels, which are the parameters of the templates that take part in the communication.

Participant type : These define the different parties that participate in the choreography. In timed automata they are processes participating in the system.

Channel types : A channel is a point of collaboration between parties, together with the specification of how the information is exchanged. As stated above, channels of WS-CDL correspond with channels of timed automata.

Variables : These are easily translated, as timed automata in UPPAAL support variables, which are used to represent some information.

Role = Template
Relation Type = Channel$^+$
Participant Type = Process$^+$
Channel Type = Channel
Variables = Variables
Choreography = Choreography$^+$ | Activity
Activity = Work Unit | Sequence | Paralelism | Choice
Sequence = Activity$^+$
Parallelism = Activity$^+$
Choice = Activity$^+$
Work Unit = State & Guard & Invariant

where the symbols +, | are BNF notation, and & is used to join information

Fig. 6. Schematic view of the translation

Now the problem is to define the behavior of each template. This behavior is defined by using the information provided by the flow of choreographies. Choreographies are sets of workunits or sets of activities. Thus, activities and workunits are the basic components of the choreographies, and they capture the behavior of each component. Activities can be obtained as result of a composition of other activities, by using sequential composition, parallelism and choice. In terms of timed automata these operators can be easily translated:

- The sequential composition of activities is translated by concatenating the corresponding timed automata.
- Parallel activities are translated by the cartesian product of the corresponding timed automata.
- Choices are translated by adding a node into the automata which is connected with the initial nodes of the alternatives.

Finally, time restrictions are associated in WS-CDL with workunits and interaction activities. These time restrictions are introduced in timed automata by means of guards and invariants. Therefore, in the event of a workunit of an activity having a time restriction we associate a guard to the edge that corresponds to the initial point of this workunit in the corresponding timed automaton.

Thus, by applying these rules we obtain three timed automata: one corresponding to the **customer** (Fig. 7), another one to the **seller** (left-hand side of Fig. 8) and the last one to the **carrier** (right-hand side of Fig. 8).

6 The Verification phase: via model checking

The model checking algorithm that UPPAAL uses is based on the symbolic model checking [3,16] that uses constraint solving. The algorithm checks if a state in a timed automata is reachable from the initial state or not. When searching the state space we need two buffers that we can call "wait" and "passed" respectively.

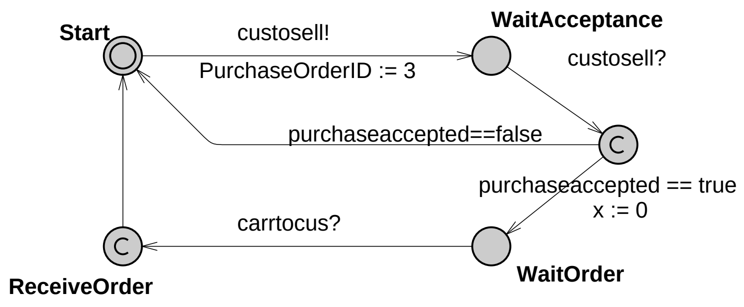


Fig. 7. The customer automaton.

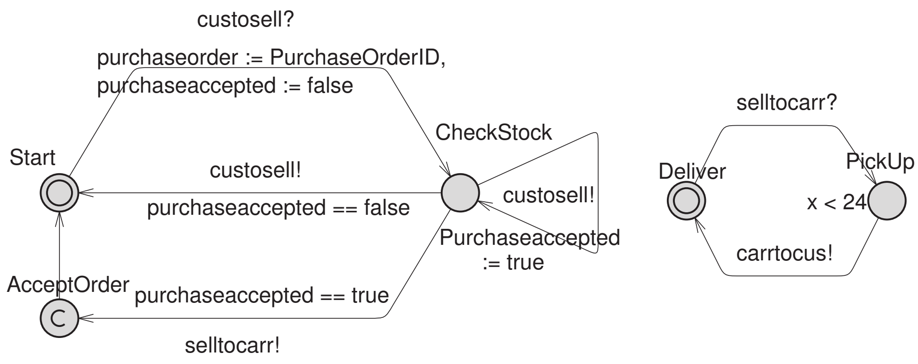


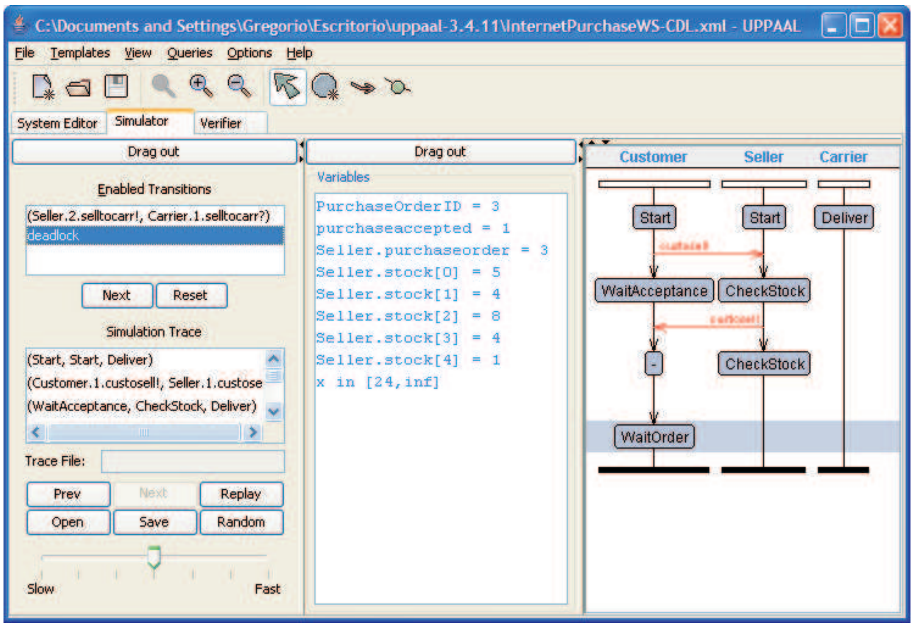
Fig. 8. Seller and carrier automata.

The wait buffer holds the states not yet explored and the passed buffer holds the states explored so far.

Algorithm 1 ***Forward Reachability Analysis***

If we do forward reachability analysis we initially store $\langle l_0, U_0 \rangle$ in the wait buffer. We then repeat the following:

1. *Pick a state $\langle l_i, U_i \rangle$ from the wait buffer.*
2. *Check if $l_i = l_f \wedge U_i \subseteq U_f$. If that is the case, return the answer yes.*
3. *If $l_i = l_j \wedge U_i \subseteq U_j$, for some $\langle l_i, U_i \rangle$ in the passed buffer, drop $\langle l_i, U_i \rangle$ and go to step 1. Otherwise save $\langle l_i, U_i \rangle$ in the passed buffer. If $U_j \subset U_i$ we can replace the state $\langle l_j, U_j \rangle$ with $\langle l_i, U_i \rangle$. (To save space)*
4. *Find all l_k that are reachable from l_i in one step regardless of the assignments, taking only actions into account. Let g_k be the set of guards on the performed transition and a_k the set of resets*
5. *Now set $U_k = reset(sp(U_i) \cap g_k, a_k)$. If $U_k \neq \emptyset$, store $\langle l_k, U_k \rangle$ in the wait buffer.*
6. *If the wait buffer is not empty go to step 1, otherwise return the answer no.*



```
(Seller.2.selltocarr!, Carrier.1.selltocarr?)
deadlock
```

```
(Start, Start, Deliver)
(Customer.1.custosell!, Seller.1.custose
(WaitAcceptance, CheckStock, Deliver)
```

```
PurchaseOrderID = 3
purchaseaccepted = 1
Seller.purchaseorder = 3
Seller.stock[0] = 5
Seller.stock[1] = 4
Seller.stock[2] = 8
Seller.stock[3] = 4
Seller.stock[4] = 1
x in [24,inf]
```

Fig. 9. The Uppaal trace for property 5

Thus, we can use the verifier of UPPAAL in order to check the properties that were identified. Notice that these properties must be adapted to consider the particular names of variables and clocks that are used in UPPAAL. For instance, the first property *"PickupOnTime"* (1) is rewritten as follows:

$$Customer.WaitOrder --> (Carrier.PickUp \wedge x < 24) \tag{5}$$

The second property, *"DeliverOnTime"* (2) is rewritten as:

$$A <> (Carrier.Deliverandx < 24) \tag{6}$$

The third property *"SatisfiedCustomer"* (3) is rewritten as follows:

$$E <> (Customer.ReceiveOrderandx < 24) \tag{7}$$

The fourth property *"EfficientService"* (4) is rewritten as follows:

$$Seller.AcceptOrder --> Customer.ReceiveOrder \tag{8}$$

Observe that the clocks $Clock_{deliver}$ is renamed to x.

We find an error in the verification of a property, concretely in Property 5 (Fig. 9). The problem appears when the seller sends the "acceptorder", but he does not send the "PickUp" message to the carrier within 24 hours. Then the carrier cannot deliver the product on time and the property is not fulfilled.

In order to correct this problem it is necessary to force the seller to send the "PickUp" message on time. For that purpose, we add an invariant to the seller state "CheckStock" labeled x < 2. With this invariant the seller must send the message within 2 hours since he has sent the message "PurchaseAccepted".

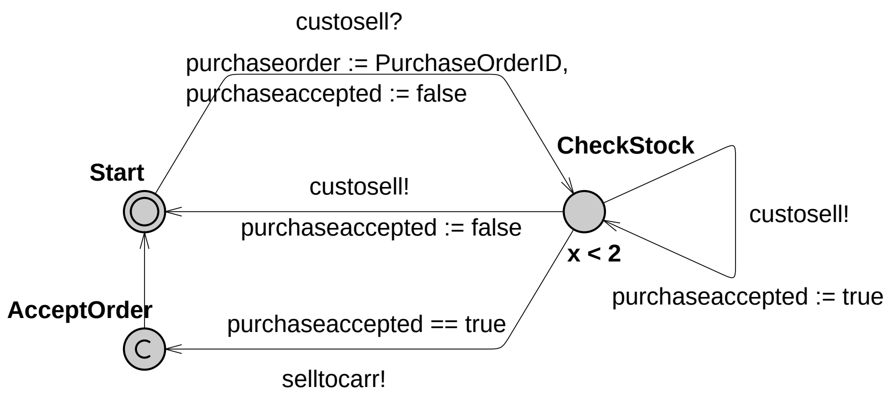


Fig. 10. Corrected Seller automaton

```
<interaction name="PickUpProductPO" channelVariable="tns:deliver-channel"
             operation="PickUpPurchaseOrder" align="true" initiate="true">
  <participate relationshipType="tns:CustIntSellCarrRS"
               fromRole="tns:Seller" toRole="tns:Carrier"/>
  <exchange name="request"
            informationType="tns:purchaseOrderType" action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
  <timeout  time-to-complete=
             "cdl:minor(cdl:getVariable("tns:Clock1","",""),"02:00")"/>?
</interaction>
```

Fig. 11. Corrected interaction

Thus, the seller automaton would be replaced with the automaton depicted in Fig. 10 and the WS-CDL interaction that represents it would be rewritten as shown in Fig. 11.

7 Conclusions and Future Work

In this work, we have presented a proposal for the analysis and verification of Web Services choreographies with time requirements. The gathering of time requirements via goal-driven diagrams, such as the KAOS extension presented in the fourth section, is a powerful tool for developing systems where time aspects determine whether the implementation presents the proper behaviors or not. However, in order to achieve this conclusion, this technique must be used together with formal specifications and formal techniques that can perform a verification process. For this purpose, the model checking technique has shown itself to be, in a wide range of systems, one of the most feasible formal method techniques.

As future work, we are working on the application of these techniques to other fields like Web Services orchestrations (WS-BPEL).

References

1. A. van Lamsweerde A. Dardenne and Stephen Fickas. Goal-directed requirements acquisition. page 350.
2. R. Alur and D. Dill. Automata for modeling real–time systems. In *In Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443. Springer–Verlag, 1990.
3. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*. Springer-Verlag, 2004.
4. Mario Bravetti, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Supporting e-commerce systems formalization with choreography languages. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 831–835, New York, NY, USA, 2005. ACM Press.
5. Mario Bravetti, Roberto Lucchi, Gianluigi Zavattaro, and Roberto Gorrieri. Web services for e-commerce: guaranteeing security access and quality of service. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 800–806, New York, NY, USA, 2004. ACM Press.
6. A. Rifaut J.F. Molderez A. van Lamsweerde C. Ponsard, P. Massonet and H. Tran Van. Early verification and validation of mission critical systems. In *Ninth International Workshop on Formal Methods for Industrial Critical Systems*, 2004.
7. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.
8. R. Darimont and A. van Lamsweerde. Formal refinement patterns for goaldriven requirements elaboration. In *Ninth International Workshop on Formal Methods for Industrial Critical Systemse, FSE-4 - 4th ACM Symp. on the Foundations of Software Engineering*, October 1996.
9. Gregorio Díaz, Fernando Cuartero, Valentín Valero Ruiz, and Fernando L. Pelayo. Automatic verification of the tls handshake protocol. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 789–794. ACM Press, 2004.
10. Gregorio Díaz, Kim Guldstrand Larsen, Juan José Pardo, Fernando Cuartero, and Valentin Valero. An approach to handle real time and probabilistic behaviors in e-commerce: validating the set protocol. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 815–820. ACM Press, 2005.
11. Gregorio Díaz, Juan José Pardo, María-Emilia Cambronero, Valentin Valero, and Fernando Cuartero. Automatic translation of ws-cdl choreographies to timed automata. In *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 230–242. Springer, 2005.
12. Nickolas Kavantzas et al. Web service choreography description language (wscdl) 1.0. http://www.w3.org/TR/ws-cdl-10/.
13. Constance Heitmeyer and Dino Mandrioli. *Formal Methods for Real-Time Computing.* John Wiley & Sons, 1996.
14. Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

15. K. Larsen, P. Pettersson, and Wang Yi. Uppaal in a nutshell. *Journal on Software Tools for Technology Transfer*, 1, 1997.
16. Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, Dec 1995.
17. Elena Navarro, Pedro Sanchez, Patricio Letelier, Juan A. Pastor, and Isidro Ramos. A goal-oriented approach for safety requirements specification. In *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06)*, pages 319–326. IEEE Computer Society, 2006.
18. Costas Courcoubetis Rajeev Alur and David L. Dill. Model-checking in dense real-time. In *Journal of Information and Computation*, 1993.
19. J. Sifakis T.A. Henzinger, X. Nicollin and S. Yovine. Symbolic model checking for real-time systems. In *In Proceedings of the IEEE Conference on Logics in Computer Science (LICS)*, 1992.
20. A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *International Conference on Software Engineering*, page 519, 2000.