A Timed Extension of WSCoL*

Luciano Baresi† Domenico Bianculli‡ Carlo Ghezzi† Sam Guinea† Paola Spoletini†

† *Dipartimento di Elettronica e Informazione*
Politecnico di Milano
{*baresi, ghezzi, guinea, spoleti*}*@elet.polimi.it*

‡ *Faculty of Informatics*
University of Lugano
domenico.bianculli@lu.unisi.ch

Abstract

Web service based applications are expected to live in dynamically evolving settings. At run-time, services may undergo changes that could modify their expected behavior. Because of such intrinsic dynamic nature, applications should be designed by adhering to the principles of design-by-contract. Run-time monitoring is needed to check that the contract between service providers and service users is fulfilled while the collaboration is in place. We describe a language to specify the expected functional and non-functional requirements that a service provider should fulfill. The language (Timed WSCoL) is a temporal extension of a previous proposal (WSCoL). We also illustrate the architecture of a run-time analyzer that checks Timed WSCoL properties. Should such properties be disproved during execution, appropriate recovery and reconfiguration actions may be put in place.

1 Introduction

A monitoring facility is an essential component of service-oriented architectures. Web services are developed and run by *service providers*, and are used by independent *users* who may compose them through workflows described in the BPEL language [1]. A systematic and sound way of designing the interactions between providers and users should follow the principles of *design-by-contract* [11]. Web service providers and users have to agree on a contract: a provider publishes a specification of what it provides (called *P-spec*) and this should be matched by the specification of the requested service (called *R-spec*) stated by the user. Both *P-specs* and *R-specs* should specify not only the functionality of the service, but also other quality attributes, such as latency, reliability, etc. Ideally, an *R-spec* and the corresponding *P-spec* should continue to match for the lifetime of the collaboration.

However, *P-spec*s may not always be available. In this case, we can still envision *R-spec*s as a way to provide defensive process design and execution. In fact, *R-spec*s may represent the properties that must hold during interaction with the external service.

In both cases, things can go wrong. For example, we can have *R-spec*/*P-spec* mismatches due to changes in the service implementation, or the quality of service, as observed by the user, may deviate from what was specified in the *R-spec*.

This is where monitoring comes in. The user must monitor the quality of observed services by identifying suitable probes for the run-time assessment of service behavior [5]. These monitoring elements are in charge of checking the behavior of the external services in their interactions with the composition.

Some of the authors did previous work on monitoring the execution of BPEL processes. A special-purpose monitoring language, called WSCoL (Web Service Constraint Language), as well as its supporting framework *Dynamo* [3] were proposed and implemented. The approach constrains BPEL processes by means of proper *monitoring rules*, which are essentially pre- and post-conditions on the interaction with external partner services. Dynamo adopts a synchronous integration of business and monitoring logics; i.e., the execution of the business process is suspended every time the framework checks the validity of a monitoring rule.

However, there are interesting properties that cannot be checked punctually. They are asynchronous in nature and require temporal operators in order to be defined. For example, we may want to express such properties as latency, availability, throughput, and reliability, which sometimes can be named collectively as *dependability* [16, 13].

In addition to what WSCoL provides, it would be useful if the user could define monitoring rules that are not limited to probing punctual interactions with external services, but

*Part of this work has been supported by the IST EU project "PLASTIC", the IST EU project "SeCSE", the italian FIRB project "ART DECO" and the italian FAR project "Discorso".

rather predicate on sequences of interactions that occur in a certain portion (*scope*) of the workflow. As an example, one might wish to specify that the average completion time of a complex workflow does not exceed a certain time bound.

To achieve these goals, WSCoL has to be extended by introducing temporal constructs. In addition, we designed the the dedicated analysis engine that can provide the required monitoring features at run-time. This paper focuses on the WSCoL extension, called *Timed WSCoL*, and on its analysis engine. The language and its use are illustrated on a running example.

The rest of the paper is organized as follows. Section 2 introduces the running example that deals with medical tele-assistance. Section 3 briefly introduces WSCoL, and discusses both its main strengths and weaknesses. Section 4 presents the new timed version of WSCoL by explaining its syntactical extensions, while Section 5 explains the modifications introduced to Dynamo to support the monitoring of such properties. Finally, Section 6 surveys related approaches, and Section 7 concludes the paper.

2. A TeleAssistance example

TA (TeleAssistance) is a composite service for medical tele-assistance of patients (see a sketch of the process in Figure 1). The service interacts with: (a) *patients* (PAs), impaired and elderly people who need basic day-by-day support; (b) a *medical laboratory* (LAB), which analyzes patients' vital parameters; (c) a *first aid squad* (FS), a team of health-care personnel who are on-call to assist patients at home during emergencies; and (d) *doctors* (MDs), who are responsible for hospitalizing patients.

A new instance of the TA service becomes active as soon as it receives a client subscription message ([RECEIVE] startAssistance), i.e., a patient enables a TA device at home. The TA process checks whether the user has subscribed to the service. If not, it replies with a negative answer and terminates; otherwise it starts a loop that waits for messages (pick activity). There are four possible kinds of messages:

- Vital parameters ([ON MESSAGE] vp). These are read at the patient's site through an appropriate device (e.g. a glucometer), and sent to the lab ([INVOKE] analyzeData). The LAB may respond with either an advice to change the drug dosage being used or with an alarmNotification. In the former case, the new dosage is notified to the patient ([INVOKE] changeDosage); in the latter, the FS is alerted with a green-code alarm ([INVOKE] Alarm(alarmNotif). Alarm notifications contain a patientID and a level which can either be mild or high) Moreover, each time the FS service is invoked, it replies with an ack message.

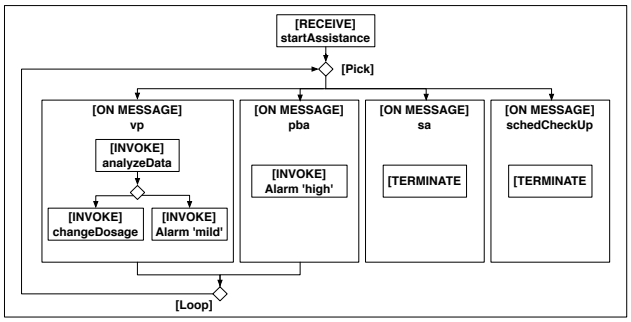

Figure 1. The Tele-Assistance service.

- A panic button alarm ([ON MESSAGE] pba). The patient, feeling ill, issues a request for immediate assistance. The TA process alerts the FS service with a red-code alarm ([INVOKE] Alarm(alarmNotif), where level is high).
- A request to stop the assistance ([ON MESSAGE sa]). The process terminates.
- a notification from doctors that the patient has been hospitalized ([ON MESSAGE schedCheckUp]). The process terminates, while the actual scheduling is performed independently by the doctors. This can be confirmed by querying the admission registry (HPR), an external service that is not part of the process.

3. WSCoL in a nutshell

WSCoL (Web Service Constraint Language) is an XML-aware assertion language (similar to JML [4]), which was originally developed to specify local pre- and post-conditions on the interactions between a BPEL process and the outside world. Due to the nature of the local conditions, this approach is synchronous, meaning that the process halts momentarily while the property is verified

A monitoring rule is composed of three parts: a *location*, a set of *monitoring parameters*, and a *monitoring expression*. The *location* consists of an XPath expression identifying an invoke, receive or pick activity within the process, and a keyword indicating whether the constraint is a pre- or a post-condition. The *monitoring parameters* are a set of meta-level data that can be used to tailor the degree of monitoring. The *monitoring expression* defines the actual constraint to be checked and is defined in WSCoL.

WSCoL provides constructs for defining both *data collection* and *data analysis*. The former is responsible for defining how monitoring data are gathered during execution. The latter defines the relationships that must hold among them.

WSCoL allows for three different kinds of data collection, which correspond to three different variable types. *Internal variables* are values obtained from the process in execution. *External variables* are values that do not belong to the process in execution, but to the outside world. Finally, *historical variables* are internal or external variables that were collected during previous activations of the monitoring framework. The collection of historical data is achieved through the use of a special-purpose WSCoL construct called *store*. This construct takes either an internal or an external variable and stores it into a persistent component. WSCoL also provides aliasing capabilities for simpler expressions. This also allows one to distinguish between variables that can be collected once (and aliased) and those that need to be collected on-the-fly everytime.

Data analysis is meant to check predicates over variables. The language allows for the use of universal and existential quantifiers, and of aggregate functions such as *max, min, avg, sum,* and *product* on collected data.

In our running example, WSCoL can be used to express interesting properties, such as: *"the vital parameters sent by the patient's glucometer must be plausible to ensure that the remote device is operating correctly"*. We could also define that: *"The lab's response to the analysis of a patient's vital parameters must be properly formatted"*.

The original version of WSCoL, however, has some limitations. Non-local, and dependability-related properties cannot be expressed. For example, the provider of the TA service might want to monitor:

- **LabServiceTime:** after sending patient's data to the lab, a reply is expected within 1 hour.
- **FSResponseTimeGreenCode:** when alerted with a green code for a certain patient, the first aid squad should send an ack to the TA within 4 hours.
- **FSResponseTimeRedCode:** when alerted with a red code for a certain patient, the first aid squad should send an ack to the TA within one hour.
- **MDCheckUp:** if the first aid squad is alerted with a red code for a certain patient three times during a time span of a week, doctors should consider hospitalizing the patient within one day.

To specify such properties we need to extend WSCoL with temporal constructs, and to appropriately modify our monitoring framework to check them.

4. Extending WSCoL with time

Temporal properties do not necessarily apply to single BPEL activities, as the original WSCoL does. In general, we need to introduce the notion of *scope*. Whenever a property is attached to a scope its evaluation occurs asynchronously with respect to the execution of the activities contained in the scope. An example is illustrated by the property **MDCheckUp**, which counts the number of occurrences of an event and correlates it to the occurrence of another event. In such a case, the two events are defined within a scope of the BPEL process.

To support the desired extension to WSCoL, the *location* field of monitoring rules now identifies the scope by an XPath expression and a special keyword `asynchronous` states that the property must be evaluated asynchronously.

4.1. Timed WSCoL

WSCoL supports propositional properties on a finite set of internal and external data. Timed WSCoL extends it with metric and non-metric temporal operators.

Timed WSCoL properties have the following syntax:

$$\phi ::= \psi \mid \neg\phi \mid \phi \wedge \phi \mid Becomes(\phi) \mid Until(\phi,\phi) \mid Between(\phi,\xi,K) \mid Within(\phi,K) \mid Count(\phi, K)$$

where ψ is a WSCoL property, and K is a positive integer number. The above syntax includes only the primitive operators, but the complete language also includes $\vee$ and $\Rightarrow$, which can be easily derived from the primitive set.

The semantics of our temporal operators are defined on timed words. Roughly speaking, a timed word is a sequence of events that occur over time (strictly monotonic), and that are time-stamped accordingly when they do. Obviously, a WSCoL property that does not contain temporal operators is evaluated at the first instant of the corresponding timed word, as are all boolean combinations of untimed properties.

On the other hand, for timed predicates and functions we must consider sub-strings or suffixes of the timed word. $Becomes$ is evaluated on two adjacent letters of a timed word. The predicate is true when its argument is true in the current letter, and false in the previous. $Until$ is evaluated on a finite suffix of the timed word, and we consider all the letters (following and including the current) until the second argument of the predicate is true. In this sub-string the first argument must always be true. $Between$ is evaluated on a finite suffix of the timed word. We drop all letters from the current onwards (included the current itself) until we find the first of the predicate's arguments. From that point on we build the suffix containing all letters that occur before K instances of time. The predicate is true if in this sub-string both arguments are true. $Within$ considers a finite sub-string containing the current and all the letters occurring before K instances of time. The predicate is true when its argument is true within this sub-string. $Count$ considers a finite sub-string containing the current and all the letters

occurring before K instances of time, and counts the number of times the argument is true within this sub-string.

Formally, the temporal extensions can be defined as follows. First, we give the definition of timed words.

Definition If E is a finite set of variable names with an associated compatible value, $e = e_1,\ e_2,\ \ldots$, with $e_i \subseteq E$, an infinite sequence of sets of data, and $t = t_1,\ t_2,\ \ldots$ an infinite sequence of time values, such that $\forall i \geq 1,\ t_i < t_{i+1}$ and $\forall t' \in \mathbb{R}^+\ \exists t_i$ in $t,\ t_i > t'$, a *timed word* is an infinite sequence $w = w_1,\ w_2,\ \ldots$ such that $w_i = (e_i,\ t_i)$.

Let $\mathbb{N}$ be the set of natural numbers, $i \in \mathbb{N}$, $w(i)$ the i^{th} element of the word w and $\varepsilon : w(i) \rightarrow e(i)$. For all timed WSCoL formulae ϕ, for all timed words $w = (e, t)$, for all $i \in \mathbb{N}$, the satisfaction relation $\models$ is defined as follows.

- if $\phi \in E, \quad w,\ i \models \phi$ iff $\phi \in \varepsilon(w(i))$;
- $w,\ i \models \neg\phi$ iff $w,\ i \not\models \phi$;
- $w,\ i \models \phi \wedge \xi$ iff $w,\ i \models \phi$ and $w,\ i \models \xi$;
- $w,\ i \models \mathit{Becomes}(\phi)$ iff $i > 0$, $w,\ i \models \phi$ and $w,\ i-1 \not\models \phi$
- $w,\ i \models \mathit{Until}(\phi, \xi)$ iff $\exists j \geq 0 \mid w,\ j \models \xi$ and $\forall k < j$, $w,\ k \models \phi$;
- $w,\ i \models \mathit{Between}(\phi, \xi, K)$ iff if $\exists j \geq 0 \mid w,\ j \models \phi$, then $\exists h \leq j + K \mid w,\ h \models \xi$
- $w,\ i \models \mathit{Within}(\phi, K)$ iff $\exists j \geq i \mid t_j - t_i < K$ and $w,\ j \models \phi$
- $w,\ i \models \mathit{Count}(\phi,\ K) = x$ iff
  - if $x = 0$ and $K \geq 0$ then $\forall j \mid t_j \leq t_i + K, w,\ j \models \neg\phi$
  - if $x = 1$ and $K \geq 0$ then $\exists j \mid t_j \leq t_i + K \wedge w,\ j \models \phi$ and $\forall h \mid t_h \leq t_i + K \wedge h \neq j,\ w,\ h \models \neg\phi$
  - if $x > 1$ and $K \geq 0$ then $\exists j \mid t_j < t_i + K \wedge w,\ j \models \mathit{Count}(\phi, t_j) = x - 1$ and $\exists h \mid t_j < t_h \leq t_i + K \wedge w,\ h \models \phi$ and $\forall l \mid t_j < t_l \leq t_i + K \wedge l \neq h,\ w,\ l \models \neg\phi$

Other temporal operators can be easily derived as follows:

- $\mathit{Sometime}(\phi) \Leftrightarrow \mathit{Until}(\mathit{true}, \phi)$
- $\mathit{Always}(\phi) \Leftrightarrow \neg\, \mathit{Sometime}(\neg\phi)$
- $\mathit{Lasts}(\phi) \Leftrightarrow \neg\, \mathit{Within}(\neg\phi)$

Since we are interested in monitoring the interaction with the outside world, a constraint in the usage of the language is that each WSCoL subformula not containing temporal operators has to be associated with an external event (typically an invocation) or with a boolean combination of them. This can be expressed either generically (e.g., indicating a generic invocation of a particular service, regardless of the number of times it is referenced in a specified scope) or with specific coordinates that identify a particular invocation in the scope. In both cases, we encode this information as a subscript to a WSCoL sub-formula (or to an entire formula). This is not needed when the untimed formula only refers to external variables.

As we have seen, the properties we define are associated with changes in the values of internal and external variables. However, not all the interesting properties can be easily associated with a variable change. For example, when monitoring the amount of time it takes an external service to respond to a call, we need additional support from the language. We need to know when an invocation is executed and when the corresponding response is received. For this purpose, we introduce the predicates $\mathit{Started}(\pi)$ and $\mathit{Finished}(\pi)$, that are respectively true when the BPEL activity uniquely identified by π (i.e., an invoke, a pick, or a receive) starts and finishes.

In order to evaluate these predicates, we enrich the timed word by collecting these "starting" and "finishing" moments. More formally, a timed word becomes an infinite sequence $w = w_1,\ w_2,\ \ldots$ such that $w_i = (e_i,\ t_i)$, where $e_i \in E \cup P$, and P is a set of variable names $\mathit{state}(\pi)$, with an associated compatible value, in which π uniquely identifies an activity within the process. These state variables can assume the following values: 1, when the activity is started by the process; 2, when the process receives the data it is waiting for; or 0 otherwise. Thanks to this enrichment, the semantics of $\mathit{Started}(\pi)$ and $\mathit{Finished}(\pi)$ can be defined as follows:

- if $\mathit{state}(\pi) \in P$, $w,\ i \models \mathit{state}(\pi)$ iff $\mathit{state}(\pi) \in \varepsilon(w(i))$;
- $w,\ i \models \mathit{Started}(\pi)$ iff $w,\ i \models \mathit{Becomes}(\mathit{state}(\pi) == 1)$;
- $w,\ i \models \mathit{Finished}(\pi)$ iff $w,\ i \models \mathit{Becomes}(\mathit{state}(\pi) == 2)$;

Therefore, in property **LabServiceTime**, the maximum amount of time (in hours) we are willing to wait for the service analyzeData to complete is expressed as:

$$\mathit{Always}(\mathit{Becomes}(\mathit{Started}(\text{[INVOKE] analyzeData})) \rightarrow \mathit{Within}(\mathit{Finished}(\text{[INVOKE] analyzeData}), 1))$$

The extensions we provide also allow us to cover the other properties presented in Section 3. **FSResponseTimeGreenCode**, for example, can be expressed as[1]:

$$\mathit{Always}(\mathit{Becomes}(\text{alarmNotif/level ==`mild'})_{\text{[INVOKE] analyzeData}} \rightarrow \mathit{Within}(\mathit{Finished}(\text{[INVOKE] Alarm}), 4))$$

In this example, we are stating that when alarmNotif/level is set to mild by the [INVOKE] analyzeData invoke activity (see subscript), the process has 4 hours to finish the [INVOKE] Alarm invoke activity.

Finally, **MDCheckUp** can be expressed as:

$$\mathit{Always}(\exists x(\mathit{Becomes}(\mathit{Count}(\text{alarmNotif/level==`high'} \wedge x\text{==alarmNotif/patientID}, 168)\text{==3})_{\text{[INVOKE] analyzeData}} \rightarrow \mathit{Within}(\texttt{returnBool}(\text{HPR, 'isInH'}, x, \text{/result})\text{==`true'}, 24)))$$

[1] **FSResponseTimeRedCode** is defined similarly.

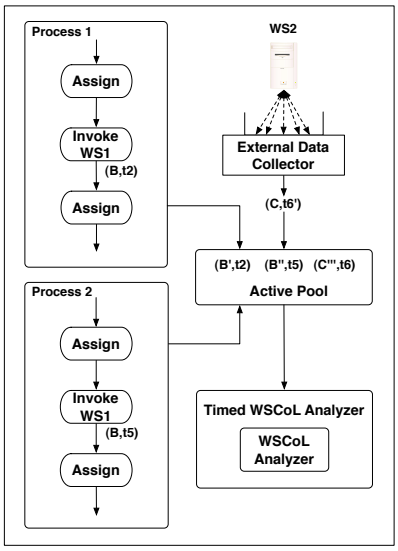

Figure 2. A run-time view of the monitoring framework.

In this example, we count the number of times the [INVOKE] analyzeData invoke activity (see subscript) receives a high alarm notification within a 168 hours time-window for patient x. As soon as it happens three times, Dynamo should notice (within 24 hours) a confirmation for the patient's hospitalization in the HPR service. To do so, we make use of the `returnBool` keyword used in WSCoL to define an external variable. It takes the following parameters: the endpoint of the HPR Web Service, the remote method to be called on the hospital's service (i.e., the isInHospital method, abbreviated as isInH), the inputs to be sent to the method (in this case the patientID), and an XPath expression used to perform data extraction on the complex message type returned by the hospital's service.

5 Monitoring timed properties

Our framework for the run-time monitoring of temporal constraints is composed of two main components. The first is an extended version of Dynamo, while the second is an external *Timed WSCoL Analyzer*. In order to understand how the two come together to ensure asynchronous monitoring, we need to briefly summarize how Dynamo achieves synchronous monitoring.

Dynamo is an AOP-based [8] extension of the ActiveBPEL orchestration engine, aimed at adding the required monitoring capabilities. ActiveBPEL uses internal tree-like representations of the process to perform execution. In these *instance trees*, each node represents an explicit (or implicit) process activity. Since all nodes implement a common interface, the engine can use the *visitor* pattern to traverse the tree and execute the process. In the previous implementation of Dynamo, we used AspectJ [7] to intercept the execution of the activities that interact with the outside world (e.g., invokes, receives, picks). The *advice* then checked a local configuration repository to see whether monitoring rules had been defined for that activity. If a constraint had to be checked, the necessary data were collected. Depending on the kind of WSCoL variable to be collected, data were either (1) extracted directly from the state of the process (associated within the engine to the corresponding instance tree) in the case of internal variables, (2) obtained by invoking external web services in the case of external variables, or (3) obtained by querying a local persistent storage component. Once all monitoring data were collected, control was momentarily passed over to the *WSCoL Analyzer* for the verification.

With respect to the previous implementation, the first main novelty is that all the data we collect need to be time-stamped and added to a special-purpose component called the *Active Pool*. The second is that, when verifying an asynchronous property, as soon as the data are collected, control is passed back to the process and not to the *WSCoL Analyzer*. The monitoring of asynchronous properties, in fact, is delegated to a component that works in parallel with the execution of the process.

Time-stamping variables. *Internal variables* are time-stamped with the execution engine's internal clock, *historical variables* are extracted from the persistent storage component and time-stamped with the *present* time, while *external variables* are slightly harder to manage.

To collect external variables, we use a special-purpose external data collector (see Figure 2) that synchronizes its clock with that of the process, and periodically queries the service for the latest value of the external variable . This value is then communicated to the *Active Pool* each time it changes. Notice that the frequency with which we poll the data source is extremely important since it can have direct consequences on the result of our monitoring. If the polling were too slow we could never become aware of a value change.

All the data placed into the *Active Pool* preserve the information regarding where they come from. Besides a time-stamp, in fact, data in the *Active Pool* have a reference to *who* modified them (i.e., which invoke call or which of the external services we are polling for external variables).

Figure 2 summarizes how variables are time-stamped and inserted into the *Active Pool*. In the example we have two instances of the same process, both communicating variable B to the *Active Pool*, and an external variable C that is obtained from a service that has nothing to do with

either process. Besides being time-stamped, the variables appear in the *Active Pool* with an superscript indicating their origin.

Asynchronous Analysis. The *Active Pool* acts as a bridge between data collection and the actual asynchronous monitoring. It is an active component which is used to "push" variable values and variable changes to our *Timed WSCoL Analyzer*. Through a simple Publish and Subscribe paradigm, the *Timed WSCoL Analyzer* can subscribe to two kinds of events. It can either (1) ask the *Active Pool* to send the value of a variable (and its time-stamp) only when it changes, through a *data-change subscription*, or (2) ask it to include a periodical variable value "push" regardless of value changes, through a *data-value subscription*.

The *Timed WSCoL Analyzer* deals with a set F of formulae to be monitored in the following way. If $|F| = n$, it has n independent components that work in parallel, each one devoted to a formula $\phi_i \in F$. Every component behaves as an automaton A_i that has to recognize if the input word, i.e. the data provided by the *Active Pool*, satisfies ϕ_i.

The automata A_i are built by exploiting the well-known correspondence between Linear Temporal Logic and Alternating Automata [9], together with variables to deal with time-stamps. To provide an informal definition of alternating automata, we need to recall the definitions of deterministic and non deterministic automata. In a deterministic (one-way) automaton, the transition function maps a $\langle$*state*, *input-symbol*$\rangle$ pair to a single state, called the next state. A configuration is a pair composed of the current state and of the unread input content. By reading the next input symbol, the automaton can reach —in a single step— a new configuration, composed of the next state and the new unread input. A configuration C may then be recursively defined to assume either the final state in which there are no more inputs, or an acceptable new configuration reachable in one step from C. The automaton accepts an input x if the configuration reached from the initial state through x is acceptable.

In a non-deterministic automaton a $\langle$*state*, *input-symbol*$\rangle$ pair is mapped onto a set of states, hence the automaton may move, by reading the next input symbol, from a configuration to a set of configurations. The traditional interpretation of non-determinism is that of *existential branching*: a non-final configuration is acceptable if, from there, there is at least one acceptable configuration which is reachable in one step. Another interpretation of non-determinism is the *universal branching*: a non-final configuration is acceptable if every configuration reachable, from there, in one step is acceptable. An alternating automaton provides both existential and universal branching. Its transition function maps a $\langle$*state*, *input-symbol*$\rangle$ pair into a boolean combination (without negation) of states. Quite naturally, $\wedge$ is used to denote universality, while $\vee$ denotes existentiality. Alternating automata may be exponentially more concise than nondeterministic automata and are very well suited for dealing with logic formulae.

We use the typical approach to translate temporal formulae into automata. In fact, we use one state for each temporal operator. The automata are then encoded directly into a net of communicating processes, one for each state of the automaton. These processes need to subscribe —in the *Active Pool*— to the variables contained in the formulae. Depending on the type of the temporal operator involved, the kind of subscription will be different. If a variable appears in an untimed sub-formula or in an unbounded temporal operator, the process will make a data-change subscription; if it appears in a bounded operator or in the $Count$ function, the subscription will be of kind data-value.

Roughly speaking, the main idea is that a bounded operator can produce its truth value through the binding specified by the operator itself. If the process is only notified when the data change, it can discover its truth value either very late or even not at all (i.e., the variable never changes its value). In the unbounded case, however, the change of value is enough to evaluate if the truth computed at the beginning is invalidated over time or not.

If a process needs to perform both data-value and data-change subscriptions, it may keep only the more complete, i.e. the data-value subscription.

Hence, the main process in the *Timed WSCoL Analyzer* representing the formula communicates both with the *Active Pool* and the processes representing the temporal subformulae, and the behavior of each of these processes is obtained by simply applying the following rules:

- If the formula is of the type $Until(\phi, \xi)$, the process checks the formula ξ at the present time, and if it is not valid, it checks ϕ. If ϕ is not valid as well, $Until(\phi, \xi)$ is false, otherwise it has to check if ξ becomes valid before ϕ changes;
- If the formula is of the type $Between(\phi, \xi, K)$, the process saves the time-stamp of the first validity occurrence of ϕ and then waits for ξ. Once ξ becomes true as well, it checks if the difference between the time-stamps is less than K. Notice that since $Between$ is bounded, a data-value subscription is performed allowing it to know in advance if the required distance between the two arguments is not respected.
- If the formula is of the type $Within(\phi, K)$, the process checks the value of ϕ and if it is not valid, it saves the present time-stamp and waits for a change in the validity of ϕ.
- If the formula is of the type $Count(\phi, K)$, the process will increment a local counting variable each time

the value of ϕ changes to true. The counting continues until the process finds, thanks to the arrival of a time-stamped variable, that more than K instances have passed since its activation. At that point, the content of the local counting variable is returned as the function's result.

All the derived operators can be easily obtained from the basic ones, hence they can be checked using similar rules.

Notice that in an alternating automaton, every time a universal branch is taken, the automaton moves into all the states that are reached by the branches, i.e., it creates multiple copies of the automaton (theoretically infinite in number). However, in our implementation, the number of processes is always bounded by the granularity of the time-stamps for bounded operators, and exactly equal to one for unbounded ones.

Consider for example **MDCheckUp** (see Figure 3 for the parsed formula). The root of the tree is the derived operator $Always$. It is implemented as a process that checks —for every letter in the timed word— if the formula represented by its subtree holds. The subtree consists of a $\Rightarrow$, meaning its right subtree is checked only if its left subtree is true.

The left subtree is an existential quantifier $\exists x$, so we consider the rest of the formula for all values of x, and verify that it is true at least once. The operator $Becomes$ monitors the changes in its subtree's truth value (i.e., from false to true) by performing data-change subscriptions to its basic variables (i.e., alarmNotif/level and alarmNotif/patientID). However, the subtree contains the temporal function $Count$, which would perform data-value subscriptions instead. Since these are more detailed than data-change subscriptions, they are preferred. The $Count$ function is implemented with a local counting variable that is incremented each time the value of the left subtree (i.e., ϕ_1) changes to true. The counting continues until the process finds, thanks to the arrival of a time-stamped variable, that more than 168 time units have passed since its activation. Notice that, since $Count$ is within an $Always$ scope, every time its left subtree (i.e., ϕ_1) becomes true a new instance of $Count$ is created. The number of these instances is finite since time is strictly monotonic and the subscription is of type data-value.

The right subtree consists of the temporal operator $Within$. If its first argument (i.e., ϕ_2) is false it continues to monitor it until either it becomes true or the difference in time-stamps between the activation time and the present becomes greater than 24 time units. The internal variables in **MDCHeckUp** are associated with the invocation [INVOKE] analyzeData.

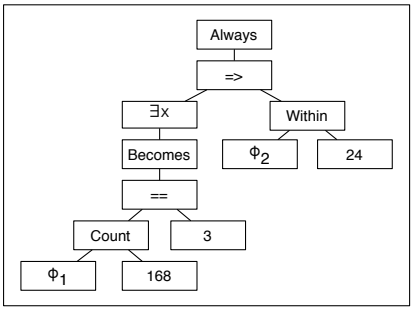

Figure 3. The syntactic tree for MDCheckUp.

6. Related work

There are several works that define specification languages for SLAs and propose appropriate monitoring architectures. In [14], Sahai et al. describe an automated and distributed SLA monitoring engine. In [6], Keller and Ludwig present the Web Service Level Agreement (WSLA) framework. It defines and monitors SLAs for web services, focusing on QoS properties such as performance and costs. Skene et al. describe a model and an analysis technique for reasoning on the monitorability of systems [15]. SLAs are expressed using the SLAng language. All these approaches focus on formally defining high-level contracts among parties (typically, between a service consumer and a service provider), hence they do not allow properties with a fine-grained granularity, such as those needed for activities in a workflow.

In [12] requirements, such as timeliness constraints, are expressed using temporal logic and KAOS, and analyzed to identify conditions under which they can be violated. If such conditions correspond to a pattern of events observable at run-time, each of them is assigned to an agent for monitoring. The approach mainly focuses on deriving monitors at design-time using a goal-driven methodology.

A framework for the run-time verification of requirements of service-based software systems is described in [10]. Requirements can be behavioral properties extracted from a composition, or assumptions on the behavior of the services used by the system, and are specified using event calculus. System events are collected at run-time and stored in an event database, while monitoring translates into an integrity constraint checking in temporal deductive databases.

Barbon et al. present an approach for monitoring BPEL service compositions [2]. Monitors can be attached to a single instance or to the whole class of process instances; they can check temporal, boolean, time-related and statistics properties, expressed in RTML, Runtime Monitoring Specification Language. Business and monitoring logics

Table 1. Comparison of the approaches

Approach	Language		Abstraction			Properties		Directives				Timeliness		
	Logic	HL/VHL	Domain	Design	Implementation	Safety	Temporal	Process	Scope	Activity	Event	Post-Mortem	Synchronous	Asynchronous
[14]		x	x				x	x						x
[6]		x	x				x	x						x
[15]		x	x				x	x						x
[12]	x		x			x	x	x				x		
[10]	x		x			x	x	x					x	x
[2]	x	x		x	x		x	x		x	x			x
Timed WSCoL	x	x	x	x	x	x	x	x	x	x	x		x	x

are kept separated by executing in parallel the monitor engine and the BPEL execution engine; code implementing monitors is automatically generated from high level specifications. With respect to our approach, they do not allow for dynamic (re-)configuration of the monitoring system in terms of rules and meta-level parameters; moreover, they don't deal with external variables, thus limiting the expressiveness of the language.

A comparison of our approach with the others mentioned above is reported in Table 1; the classification follows the taxonomy proposed in [5].

7. Conclusions and future work

The paper presented Timed WSCoL, which extends WSCoL with temporal operators. Besides the new operators, the addition of temporal properties also requires the shift to asynchronous monitoring. Properties are no more local but span over different interactions with partner services. The paper also presented the machinery needed to check these properties, and exemplified the approach on a simple case study. The proposed extension allows a more accurate supervision of composite services, and makes Dynamo a complete and integrated solution for monitoring the execution of BPEL processes.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1, May 2003.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Runtime monitoring of instances and classes of web service compositions. In *ICWS '06 Proceedings*, pages 63–71. IEEE Computer Society, 2006.

[3] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *ICSOC 2005 Proceedings*, volume 3826 of *LNCS*, pages 269–282. Springer, 2005.

[4] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[5] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, 2004.

[6] A. Keller and H. Ludwig. Defining and monitoring service-level agreements for dynamic e-business. In *Lisa '02 Proceedings*, pages 189–204. USENIX Association, 2002.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 Proceedings*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP 1997 Proceedings*, volume 1242 of *LNCS*, pages 220–242. Springer, 1997.

[9] O. Kupferman and M. Vardi. Weak alternating automata are not that weak. In *ISTCS'97 Proceedings*, pages 147–158. IEEE Computer Society Press, 1997.

[10] K. Mahbub and G. Spanoudakis. A framework for requirents monitoring of service based systems. In *ICSOC '04 Proceedings*, pages 84–93. ACM Press, 2004.

[11] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[12] W. N. Robinson. Monitoring web service requirements. In *RE'03 Proceedings*, pages 65–74. IEEE Computer Society, 2003.

[13] B. Sabata, S. Chatterjee, M. Davis, J. J. Sydir, and T. F. Lawrence. Taxomomy of QoS specifications. In *WORDS '97 Proceedings*, pages 100–107. IEEE Computer Society, 1997.

[14] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, and F. Casati. Automated SLA monitoring for web services. In *DSOM '02 Proceedings*, volume 2506 of *LNCS*, pages 28–41. Springer, 2002.

[15] J. Skene, A. Skene, J. Crampton, and W. Emmerich. The monitorability of service-level agreements for application-service provision. In *WOSP '07 Proceedings*, pages 3–14. ACM Press, 2007.

[16] H.-L. Truong, R. Samborski, and T. Fahringer. Towards a framework for monitoring and analyzing QoS metrics of grid services. In *E-SCIENCE -06 Proceedings*, pages 65–72. IEEE Computer Society, 2006.