Fundamenta Informaticae XXI (2009) 1001–1017
IOS Press

Runtime monitoring of contract regulated web services

Alessio Lomuscio
Department of Computing
Imperial College London, UK

Wojciech Penczek
Institute of Computer Science, PAS, PL, and
University of Humanities and Science, Siedlce, PL

Monika Solanki
Department of Computer Science
University of Leicester, UK

Maciej Szreter
Institute of Computer Science, PAS, PL

Abstract. We investigate the problem of locally monitoring contract regulated behaviours in agent-based web services. We encode contract clauses in service specifications by using extended timed automata. We propose a *non intrusive* local monitoring framework along with an API to monitor the fulfillment (or violation) of contractual obligations. A key feature of the framework is that it is fully symbolic thereby providing a scalable solution to monitoring. At runtime execution steps generated by the service are passed as input to the runtime monitor. Conformance of the execution against the service specification is checked using a symbolically represented extended timed automaton. This allows us to monitor service behaviours over large state spaces generated by multiple, long running contracts. We illustrate our methodology by monitoring a service composition scenario from the vehicle repair domain, and report on the experimental results.

1. Introduction

Web services (WS) are now considered one of the key technologies for building new generations of digital business systems. Industrial strength distributed applications can be built across organisational boundaries using services as basic building blocks. According to a largely widespread view services are implemented by *agents* acting socially in the system and the environment to maximise their own utility [26]. One of the main advantages of building distributed systems as agent-based web services is the flexibility they provide in terms of composition. However, when services are combined a significant challenge is to provide effective mechanisms to regulate their interactions. This is a well-known problem in multi-agent systems (MAS) research where a variety of concepts such as norms [9], institutionalised power [8] and commitments [6] have been studied to regulate and predict the behaviour of large MAS in rich human like context.

In WS the traditional notion employed for similar purposes is the one of service level agreement (SLA) [12]. They provide a useful mechanism to establish agreed levels of service provision when interactions are invoked within certain parameters. Although SLAs are useful, they can represent only basic agreements of service provision. Applications running complex, human-like activities require more general and sophisticated declarative specifications certifying legal-like agreements among the parties. Indeed, in an environment where previously unknown services are dynamically discovered and binded, their composition is required to be underpinned by binding agreements. Additionally, agents maximising their own utilities may indeed choose to violate these agreements for a better payoff differently. While this is unavoidable, we may still wish to monitor the executions and track at run time the agreements that are being violated.

A useful concept from the legal domain in this sense is the one of *contract* as found in human societies. Should a contract be broken by one of the parties, additional rights and/or obligations (e.g., penalties to be paid) may be applicable to some party. Contracts may not simply prescribe certain actions depending on certain states, but may go as far as to specify timing constraints (e.g, deadlines), or more sophisticated measures (such as the number of actions per temporal interval, as in some QoS (Quality of service) agreements).

In this paper, we study the problem of monitoring runtime behaviours of *contract regulated web services*. While contracts are usually negotiated offline, it is of interest to monitor at runtime whether interactions between WS are complying to the contracts stipulated between the parties. Runtime monitoring of web services is concerned with the actual, rather than possible state transitions occurring in the system. A runtime monitor continuously checks the executions against a model of correct behaviour previously encoded. In the case of contract-based web services we are interested in monitoring at runtime whether the contracts the web services are supposed to adhere to are violated in a given run of the system, and if so, what action follows from this behaviour, e.g, whether recovery actions are performed.

Monitoring complex interactions such as the ones above is non-trivial. The key issue is the one of scalability. It is relatively easy to envisage a methodology whereby contracts and possible behaviours are explicitly stored in memory and the stream of events at local web-service level is matched at runtime against the envisaged contract-compliant runs. However with many complex contracts to be verified and several WS present in a system the present approach is unlikely to be effective in any scenario where the range of possible behaviours is large.

In this paper we put forward a "symbolic" solution to the problem above. We represent both all possible behaviours and the contractually-correct ones as an appropriate timed automata [1] at local web service level. Specifically we present a local contract runtime monitor (CRM) based on the symbolic toolkit Verics [5], a symbolic model checker for timed-automata. CRM checks the local service's execution at runtime against the symbolic representations provided, and reports back to the service (or directly to the engineer) any mismatch, or *violation*, between the contract-compliant behaviours originally prescribed and the ones actually received in the input stream. Note that differently from other lines of research we do not wish to monitor the overall service composition here. Instead we focus on a single service and aim to monitor continuously its executions, i.e., the change of its local variables and actions. This is of relevance to several application areas where individual enterprises wish to monitor whether any of their executions violates existing contracts, or service level agreements.

The significant advantage of the approach is that we do not need to keep the whole state space of the possible and the contract-compliant behaviours in memory but we can simply call the timed-automata engine at runtime to match moves against the stream of events coming from the input. Because of the requirements of the setting the approach extends conventional timed automata with additional constraints to allow the specification of compliance and violation of contracts. As discussed below the memory footprint of the CRM is also very attractive as is its performance. Additionally timed-automata offer us a natural formalism to work with any timing properties of interest. The approach is also inherently scalable as it enables us to monitor in parallel several independent contracts, or several independent clauses in a single contract.

The rest of the paper is structured as follows: in Section 2 we briefly introduce the formalism of timed automata as used here. Section 3 presents our monitoring framework. We analyse a motivating case study in 4 and discuss the monitoring results. Section 5 presents related work and conclusions.

2. Monitoring via Timed Automata

Let $\mathbb{N}$ denote the set of naturals (including 0), $\mathbb{Z}$ - the set of integers, $\mathbb{Q}$ - the set of rational numbers, and $\mathbb{R}$ ($\mathbb{R}_+$) - the set of (non-negative) reals.

2.1. Variables and Clocks

Let V be a finite set of integer variables. The set of *arithmetic expressions* over V, denoted $Ex(V)$, is defined by the following grammar: $ex ::= c \mid v \mid v \otimes c \mid c \otimes v \mid v \otimes v$, where $c \in \mathbb{Z}$, $v \in V$, and $\otimes \in \{+, -\}$.
The set of *boolean expressions* over V, denoted $Bool(V)$, is defined by $\beta ::= true \mid ex \sim ex \mid \beta \wedge \beta \mid \beta \vee \beta \mid \neg\beta \mid (\beta)$, where $ex \in Ex(V)$ and $\sim \in \{=, \neq, <, \leq, \geq, >\}$.
The set of *instructions* over V, denoted $Ins(V)$, is given by $\alpha ::= \epsilon \mid v := ex \mid \alpha\alpha$, where $v \in V$, $ex \in Ex(V)$.

Moreover, by $Ins^L(V)$ we denote the subset of $Ins(V)$ such that for each $\alpha = \alpha_1 \dots \alpha_m \in Ins^L(V)$, where $\alpha_i = (v_i := ex_i)$ for $1 \leq i \leq m$, we have $\bigcap_{i=1}^{m}\{v_i\} = \emptyset$, i.e., each v_i is assigned a new value in α at most once.

By a *variable valuation* we mean any total mapping $\mathbf{v} : V \longrightarrow \mathbb{N}$. We extend the mapping $\mathbf{v}$ to expressions of $Ex(V)$ in the usual way. The satisfaction relation ($\models$) for the boolean expressions is also standard.

Given a variable valuation $\mathbf{v}$ and an instruction $\alpha \in Ins^L(V)$, we denote by $\mathbf{v}_u(\alpha)$ the valuation $\mathbf{v}'$, obtained after executing α at $\mathbf{v}$ (updating a valuation), which is formally defined as follows:

- if $\alpha = \epsilon$ then $\mathbf{v}' = \mathbf{v}$,
- if $\alpha = (v := ex)$, then $\mathbf{v}'(v) = \mathbf{v}(ex)$ and $\mathbf{v}'(v') = \mathbf{v}(v')$ for all $v' \in V \setminus \{v\}$,
- if $\alpha = \alpha_1\alpha_2$, then $\mathbf{v}_u{}' = (\mathbf{v}_u(\alpha_1))(\alpha_2)$.

Let $\mathcal{X} = \{x_1, \dots, x_{n_\mathcal{X}}\}$ be a finite set of real-valued variables, called *clocks*. The set of *clock constraints* over $\mathcal{X}$ and V, denoted $\mathcal{C}(\mathcal{X}, V)$, is defined by the grammar: $\mathfrak{cc} ::= true \mid x_i \sim c \mid x_i \otimes x_j \sim c \mid x_i \otimes x_j \sim v \mid x_i \otimes v \sim c \mid v \otimes w \sim x_i \mid \mathfrak{cc} \wedge \mathfrak{cc}$, where $x_i, x_j \in \mathcal{X}$, $v, w \in V$, $c \in \mathbb{N}$, $\otimes \in \{+, -\}$, and $\sim \in \{\leq, <, =, >, \geq\}$. Let $\mathcal{X}^+$ denote the set $\mathcal{X} \cup \{x_0\}$, where $x_0 \notin \mathcal{X}$ is a fictitious clock representing the constant 0. A *clock-to-clock assignment* A over $\mathcal{X}$ is a function $A : \mathcal{X} \longrightarrow \mathcal{X}^+$. $Asg(\mathcal{X})$ denotes the set of all the assignments over $\mathcal{X}$. By a *clock valuation* we mean a total mapping $\mathbf{c} : \mathcal{X} \longrightarrow \mathbb{R}_+$. The satisfaction relation ($\models$) for a clock constraint $\mathfrak{cc} \in \mathcal{C}(\mathcal{X}, V)$ under a clock valuation $\mathbf{c}$ and a variable valuation $\mathbf{v}$ is defined as:

- $(\mathbf{c}, \mathbf{v}) \models (x_i \otimes v \sim c)$ iff $\mathbf{c}(x_i) \otimes \mathbf{v}(v) \sim c$,
- the other cases are defined similarly.

In what follows, the set of all the pairs $(\mathbf{c}, \mathbf{v})$, composed of a clock and a variable valuation, satisfying a clock constraint $\mathfrak{cc}$ is denoted by $[\![\mathfrak{cc}]\!]$. Given a clock valuation $\mathbf{c}$ and $\delta \in \mathbb{R}_+$, by $\mathbf{c} + \delta$ we denote the clock valuation $\mathbf{c}'$ such that $\mathbf{c}'(x) = \mathbf{c}(x) + \delta$ for all $x \in \mathcal{X}$. Moreover, for a clock valuation $\mathbf{c}$ and an assignment $A \in Asg(\mathcal{X})$, by $\mathbf{c}(A)$ we denote the clock valuation $\mathbf{c}'$ such that for all $x \in \mathcal{X}$ we have $\mathbf{c}'(x) = \mathbf{c}(A(x))$ if $A(x) \in \mathcal{X}$, and $\mathbf{c}'(x) = 0$ if $A(x) = x_0$. Finally, by $\mathbf{c}^0$ we denote the *initial* clock valuation, i.e., the valuation such that $\mathbf{c}^0(x) = 0$ for all $x \in \mathcal{X}$.

2.2. Timed Automata with Discrete Data

In this paper we assume a slightly modified definition of timed automata with discrete data [27], which extend the standard timed automata of Alur and Dill in the following way:

Definition 2.1. A *timed automaton with discrete data* (TADD) is a tuple $\mathcal{A} = (\Sigma, L, l^0, V, \mathcal{X}, \mathcal{E}, \mathcal{I})$, where

- Σ is a finite set of *labels (actions)*,
- L is a finite set of *locations*,
- $l^0 \in L$ is the *initial location*,
- V is the finite set of integer variables,
- $\mathcal{X}$ is the finite set of clocks,
- $\mathcal{E} \subseteq L \times \Sigma \times Bool(V) \times \mathcal{C}(\mathcal{X}, V) \times Ins^L(V) \times Asg(\mathcal{X}) \times L$ is a *transition relation*, and
- $\mathcal{I} : L \longrightarrow \mathcal{C}(\mathcal{X}, \emptyset)$ is an *invariant function*.

The invariant function assigns to each location a clock constraint (without integer variables[1]) expressing the condition under which $\mathcal{A}$ can stay in this location. Each element $t = (l, a, \beta, \mathfrak{cc}, \alpha, A, l') \in \mathcal{E}$ denotes a transition from the location l to the location l', where a is the label of the transition t, β and $\mathfrak{cc}$ define the enabling conditions for t, α is an instruction to be performed, and A is a clock assignment. Moreover, for a transition $t = (l, a, \beta, \mathfrak{cc}, \alpha, A, l') \in \mathcal{E}$ we write $source(t)$, $label(t)$, $v\_guard(t)$, $cv\_guard(t)$, $instr(t)$, $asgn(t)$ and $target(t)$ for l, a, β, $\mathfrak{cc}$, α, A and l', respectively.

An example of a TADD can be found in Section 4.1. The automaton in Figure 5 is composed of 4 locations and 3 transitions, where $s4$ is the initial location. Labels on the transitions are, e.g., $SendVehicle!$ and $SendAssessed!$. Invariant is, e.g., $x <= 7$ where x is the clock. Reset on clock x is defined as $x = 0$ on the transitions. The semantics of a TADD $\mathcal{A}$ is given below:

Definition 2.2. The *semantics* of $\mathcal{A} = (\Sigma, L, l^0, V, \mathcal{X}, \mathcal{E}, \mathcal{I})$ for an initial variable valuation $\mathbf{v}^0 : V \longrightarrow \mathbb{Z}$ is a labelled transition system $\mathcal{S}(\mathcal{A}) = (Q, q^0, \Sigma_\mathcal{S}, \longrightarrow)$, where:

- $Q = \{(l, \mathbf{v}, \mathbf{c}) \mid l \in L \wedge \mathbf{v} \in \mathbb{Z}^{|V|} \wedge \mathbf{c} \in \mathbb{R}_+^{|\mathcal{X}|} \wedge \mathbf{c} \models \mathcal{I}(l)\}$ is the set of states,
- $q^0 = (l^0, \mathbf{v}^0, \mathbf{c}^0)$ is the initial state,
- $\Sigma_\mathcal{S} = \Sigma \cup \mathbb{R}_+$ is the set of labels,
- $\longrightarrow \subseteq Q \times \Sigma_\mathcal{S} \times Q$ is the smallest transition relation such that:

[1]To ensure the monotonicity of the timed successor relation.

– for $a \in \Sigma$,
$(l, \mathbf{v}, \mathbf{c}) \xrightarrow{a} (l', \mathbf{v}', \mathbf{c}')$ iff there exists a transition $t = (l, a, \beta, \mathfrak{cc}, \alpha, A, l') \in \mathcal{E}$ such that $\mathbf{v} \models \beta$, $(\mathbf{c}, \mathbf{v}) \models \mathfrak{cc}$, $\mathbf{v}' = \mathbf{v}_u(\alpha)$, $\mathbf{c} \models \mathcal{I}(l)$, and[2] $\mathbf{c}' = \mathbf{c}(A) \models \mathcal{I}(l')$ *(action transition)*,

– for $\delta \in \mathbb{R}_+$,
$(l, \mathbf{v}, \mathbf{c}) \xrightarrow{\delta} (l, \mathbf{v}, \mathbf{c} + \delta)$ iff $\mathbf{c} \models \mathcal{I}(l)$ and $\mathbf{c} + \delta \models \mathcal{I}(l)$ *(time transition)*.

A transition $t \in \mathcal{E}$ is *enabled* at a state $(l, \mathbf{v}, \mathbf{c})$ if $\mathbf{v} \models v\_guard(t)$, $(\mathbf{c}, \mathbf{v}) \models cv\_guard(t)$ and $\mathbf{c}(asgn(t)) \models \mathcal{I}(target(t))$. Intuitively, in the initial state all the variables are set to their initial values, and all the clocks are set to zero. Then, at a state $q = (l, \mathbf{v}, \mathbf{c})$ the system can either execute an enabled transition t and move to the state $q' = (l', \mathbf{v}', \mathbf{c}')$, where $l' = target(t)$, the valuation of the variables is changed according to $instr(t)$, and the clock valuation is changed according to $asgn(t)$, or move to the state $q' = (l, \mathbf{v}, \mathbf{c} + \delta)$ which results from passing some time $\delta \in \mathbb{R}_+$ such that $\mathbf{c} + \delta \models \mathcal{I}(l)$.

2.3. TADD Semantics for contract monitoring

In our framework, we use the TADD for an agent-based service as a formal structure for the specification of all possible behaviours (contract-compliant or otherwise). Inspired by related work in the formal representation of states of compliance and violation [15], we partition the set of global states Q of $\mathcal{S}(\mathcal{A})$ for $\mathcal{A} = (\Sigma, L, l^0, V, \mathcal{X}, \mathcal{E}, \mathcal{I})$ into two subsets G and R such that $G \cap R = \emptyset$[3]. The set G represents *green* (or *ideal*) states, whereas R represents the *red* (or *non-ideal*) ones. Intuitively, G contains the states of compliance and R contains the states of violation with respect to the contract, i.e., the whole set of clauses being included. Figure 1 illustrates the intuition behind the semantics.

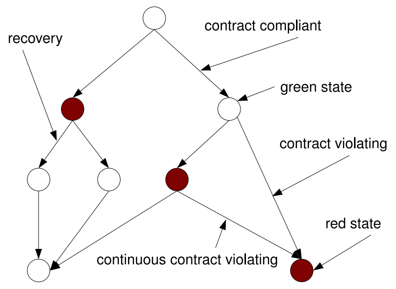

Figure 1. Partitioning of states and transitions in a TADD

Based on the above partitioning each action transition (q, a, q') of $\mathcal{S}(\mathcal{A})$ can be one of the following four types of transitions:

- **Contract compliant**: between green and green states, i.e., $q, q' \in G$. These transitions occur when the observed behaviour is in compliance with the prescribed behaviour of the contract.

- **Contract violating**: between green and red states, i.e., $q \in G$ and $q' \in R$. These transitions occur when the observed behaviour violates the prescribed behaviour of the contract.

[2]Note that satisfaction of invariants is ensured by the definition of Q.
[3]This partition is obtained "location-wise" from a partition of the set of locations L.

- **Recovery**: between red and green states, i.e., $q \in R$ and $q' \in G$. These transitions occur when a recovery action is taken by the service after a violation of the prescribed behaviour is recorded.

- **Continuous contract violating**: between red and red states, i.e., $q, q' \in R$. The transitions occur when no recovery results from a previous violation.

We say that there is a *step* from state q_1 to q_2 in $\mathcal{A}$ if $q_1 \xrightarrow{\delta_1} q_1' \xrightarrow{a} q_2' \xrightarrow{\delta_2} q_2$, for some states $q_1', q_2' \in Q$, $\delta_1, \delta_2 \in \mathbb{R}_+$, and $a \in \Sigma$.

2.4. Querying of TADD for monitoring contracts

In order to monitor contracts at runtime, we monitor the variation in the variables of the current state of the service. We query the symbolic representation, i.e., the TADD, by inputting the previously observed state and the currently observed state. We check whether there is a transition in the formally specified TADD representation between the two states.

In our approach, we rephrase the problem of local monitoring of executions against contract compliant behaviours into the following model checking problem: for a given TADD $\mathcal{A}$ and a pair (Q, Q') of sets of global states of $\mathcal{S}(\mathcal{A})$, we check whether there are two states $q \in Q$ and $q' \in Q'$ such that there is a step from q to q'. If so, we denote the step as $Q \rightsquigarrow Q'$. We use this operation as follows: first we check if there is a transition from the source set Q to the subset of target states Q' being the red states R (formally: $Q \rightsquigarrow (Q' \cap R)$). If so, then *non compliance* (RED) is reported. If there is no such a step, we check if $Q \rightsquigarrow Q'$. If the result is positive we report *compliance* (GREEN), or INVALID TRANSITION in the other case.

Technically, the transition relation is first encoded into a propositional formula. Then for each step, this propositional formula is conjuncted with the encodings of a pair of sets of states specified above, with the conjunction of formulas encoding sets implementing the set union. The satisfiability of the resulting formula is tested. A transition exists when the formula is satisfiable, and does not exist otherwise.

Thus we check steps of one transition in length that can occur anywhere in the system. Note that the fact that we check one step at a time does not mean that we deal with single-step contract. We benefit by using SAT because we can encode the whole system of all possible behaviours very efficiently, and that queries are efficiently answered. Our tool uses MiniSAT [7] for checking satisfiability, but any standard SAT-solver capable of processing propositional formulas in the conjunctive normal form can be applied.

3. Runtime monitoring framework

Our approach for run-time monitoring of contract-based web services (RMCWS), is illustrated in Figure 2. For each agent to be monitored all its possible behaviours (contract-compliant and otherwise) are represented as a TADD and stored in the checker. At regular intervals (whose granularity is smaller than the smallest possible sequence of local transitions, typically several per second), execution snapshots taken at runtime are passed to RMCWS as inputs. The BMC based monitoring engine checks the snapshots against their TADD specification and reports back to RMCWS whether the actual runtime behaviours are in compliance with the contractually prescribed behaviour as specified in the TADD, or, if not, states the clause that has been violated in the present transition. In this section we enumerate and discuss the core components of RMCWS followed by a detailed discussion on the monitoring mechanism.

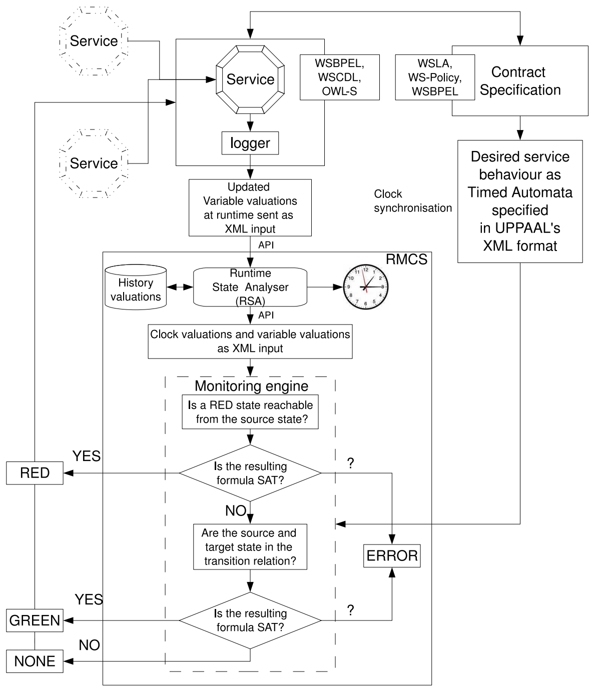
Service
Service
WSBPEL, WSCDL, OWL-S
WSLA, WS-Policy, WSBPEL
Contract Specification
Service
logger
Desired service behaviour as Timed Automata specified in UPPAAL's XML format
Updated Variable valuations at runtime sent as XML input
Clock synchronisation
API
RMCS
History valuations
Runtime State Analyser (RSA)
API
Clock valuations and variable valuations as XML input
Monitoring engine
Is a RED state reachable from the source state?
YES
RED
Is the resulting formula SAT?
?
NO
Are the source and target state in the transition relation?
ERROR
YES
GREEN
Is the resulting formula SAT?
?
NO
NONE

Figure 2. The general architecture and methodology

3.1. Runtime architecture

Agents implementing WS are the primary entities within our framework. Service behaviour and contracts associated with them may be specified at a high level using WS standards, e.g., WSBPEL [21] and contracts, e.g., WSLA [12]. The TADD specification for the service is engineered from these interface representations.

A significant feature of our framework is that we do not place any restriction on service implementation in terms of development infrastructure and execution platforms. Central to our framework is a *non intrusive* approach to monitoring. The mechanism works independently of service execution.

The module responsible for linking the service to the monitoring mechanism is the " logging framework". Each service to be monitored is associated with a logger. The logger records a "snapshot" of the variables of interest that are to be monitored. Snapshots may be finely grained, i.e., every change in valuation is recorded or coarse, i.e, recorded after every pre-specified or random number of changes. Snapshots may also be time bound, i.e., taken after a specific time interval. Each snapshot captures variable valuation as they are generated, updated by the service or received from partners. Every snapshot is passed to the runtime state analyser using a dedicated API provided by the logging framework.

TADDs for services: The specification of service behaviour used by RMCWS is a TADD representation as in Section 2. We use the XML format generated by the model checker UPPAAL [22] for representing the TADD. Our choice is motivated by the fact that UPPAAL provides a user friendly GUI. This is of great help to system engineers when modelling the TADDs. Secondly, the XML representation format can be modified easily in order to take into account any extensions to the TADD model. As illustrated

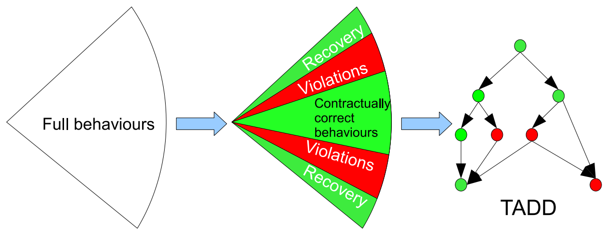

Figure 3. Set of behaviours for a service

in Figure 3, the TADD specification encodes all possible desired behaviours for a service. Typically, the full set of behaviours for a contract regulated service can be derived from:

- its contractually compliant behaviours. These behaviours encapsulate contractual obligations for the service.
- behaviours that are classified as violations of the contract.
- behaviours that define a recovery from incurred violations.

There is a one-to-one correspondence between variables defined in the TADD and the service implementation in terms of types and names i.e., variables names and their types across the two representations are kept identical for simplicity. The logging framework passes execution snapshots to the analyser component of the engine as an XML data structure. One such snapshot is illustrated in Listing 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<VarValuations>
 <var name="receivedPO">true</var>
 <var name="numberOfDays">5</var>
</VarValuations>
```

Listing 1. Snapshot from logger to RSA

Runtime State Analyser (RSA): The runtime state analyser interfaces with the logger for receiving snapshots of latest variable valuations generated by the service. Snapshots are passed to the RSA via the logging framework. RSA is also responsible for updating clocks by querying the system hardware, in accordance with the granularity of a *tick* chosen by the service. A tick can be defined in terms of seconds, minutes, hours or days i.e., clock values may be captured every second, minute or day or any other interval chosen by the service. Clocks may also be updated based on resets and assignments defined in the TADD for the service. The monitoring engine reports back any resets or assignments made to the clocks, along with reporting the results, e.g., if the monitoring engine reports that a clock x is reset, the current valuation of the clock is discarded and the clock is re-initialised. Clock resets and updates are significant especially when a recovery action is taken against a violation of contract. In such scenarios one would like to start the monitoring again with clock valuation recorded before the violation occurred. The clock valuations once recorded are then added to the variable valuation snapshot received from the logger. RSA is also responsible for storing the history of service executions and passing the augmented snapshots to the monitoring engine.

The runtime information passed to the monitoring engine from the RSA consists of one or several steps. A step is a pair of consecutive snapshots, represented as “source” and “target” states. The states define immediately previous (source) and current (target) clock and variable valuations recorded for the service. An example step for the case study in section 4, is illustrated in Listing 2 below.

```
<model_file>Repair.xml</model_file>
<step>
 <source>
  <component name="RepairCompany">
   <clock name="x">3</clock>
  </component>
  <var name="maxRepairRequestTime">7</var>
 </source>
 <target>
  <component name="RepairCompnay">
   <clock name="x">5</clock>
  </component>
 </target>
</step>
```

Listing 2. Snapshot from RSA to the monitoring engine

Any component of a source or a target such as a clock valuation, or a variable valuation can be omitted. Thus each set can range from containing only the system states (no values given at all) to a single state (every component is specified).

3.2. The monitoring engine

The monitoring engine is the core component responsible for testing the conformance of runtime service behaviour presented as an input from the RSA, against the prescribed TADD specification of the service.

Each execution step passed to the engine is encoded and its conformance to the TADD specification is tested by means of the model checking approach described in Section 2.4. Our SAT-based verification method does not need to construct the complete model for $\mathcal{A}$, which could be unfeasible for both the explicit-state [10] and BDD-based methods [17]. Instead the timed automaton is encoded as a propositional formula, but testing of its satisfiability is postponed until the concrete source and target states of an execution step are provided. This significantly reduces the computational cost as information about concrete states prunes the state space to be searched.

The engine monitors if the service has taken an execution step from the source set to the target set of states in accordance with its prescribed TADD. In addition, it checks if it is possible to reach a target red state from a given set of source states. In the general case the system consists of several components represented by automata; if at least one component of a location reachable as a result of the transition is red, then this fact is reported.

Monitoring results: The engine checks at runtime whether the stream of execution steps received as inputs from the RSA, conforms with its symbolic representation of all possible behaviours. For each execution step, the answer returned by the monitoring engine is one of the following:

- **GREEN -** This represents the fact that the step is conforming with the specification, i.e., there is a contract compliant transition between the source and target states.

- **RED -** This represents the fact that a red state is reached as a target of the transition given, i.e., a contract has been violated as a result of the transition. This also signifies the fact that the inputs do not comply with the extended format of the TADD for the service.
- **INVALID TRANSITION -** This represents the fact that the step does not conform to the specification, i.e., there is no such transition.
- **ERROR -** This represents the fact that the specification is incorrect; for example syntax errors are detected, or undefined variables occur at specified locations.

Results reported at runtime may be analysed in several ways. In case of contract compliant transitions, the service can continue executing as per the orchestrated workflow. For contract violating transitions, the service administrator may impose on the service to execute one of the prescribed recovery transition. In other cases the administrator may choose to override the violations reported and allow the service to continue execution. For a continuous contract violating transition being reported, the service may be stopped. Finally, the outputs generated may be stored in a log file for future offline analysis.

4. A vehicle repair contract: case study

We now present a description of a case study – the vehicle repair, which is a web-service coming from an industrial usecase in an actual project. This is followed by a detailed discussion on the local monitoring and analysis of one of the agents in the composition.

We consider a service composition scenario that defines a repair contract between a client (C) and a vehicle repair company (RC). Communication between C and RC is facilitated via web service interfaces. A repair contract specifies details concerning a particular repair, i.e., the type of repair to be performed, price, dates, pickup and delivery locations etc. For simplicity we only model the behaviour of RC. Table 1 identifies some of the contract clauses governing the actions taken by RC, the deadlines against which the contracts are monitored, if the clause can be violated, and, if a violation is recorded, whether any recovery is possible. Note that in some cases RC may take an "offline" action, in response to a violation from which no recovery may be possible. For example consider clause 6: "For any violation take recovery action within $maxRecoveryTime$ - number of days". If the recovery action is not taken, C may take an offline legal action against RC.

The informal behaviour of RC is described as follows. When RC receives a request from C to undertake a repair job, it sends a repair proposal. In response, C sends an acceptance or rejection message. If accepted, RC sends a contract initiation message to C. RC then waits for the vehicle to arrive, failing which it sends two reminders to C. If the vehicle fails to arrive, it takes an offline action. As per the contract, RC is *obliged* to assess the damage, repair the vehicle and send a report to C. On receiving the report, C is *obliged* to send payment to RC. If the payment is not sent, RC sends two reminders to C and then takes an offline action.

The actions taken by RC in response to messages sent by C are monitored to meet the deadlines set for various activities as per the contract. Failure to meet deadlines is considered a violation of the contractual obligations. In some cases a recovery from the violation may be possible.

We assume the contract has been negotiated offline and obligations, defined in terms of their respective contract clauses, have been agreed by each of the contract parties.

clause	Contract regulated actions	Deadline	Violation	Recovery
1	Receives a repair request by C	5 days	-	-
2	Sends a repair proposal to C	7 days	-	-
3	Assess damage to the vehicle	3 days	yes	yes
4	Execute repair	30 days	yes	yes
5	Send repair report to C	5 days	yes	yes
6	For any violation take recovery action	3 days	yes	no (take offline action)

Table 1. Some contract regulated actions for RC

4.1. Monitoring the runtime behaviour of the Repair Company

The full set of behaviours of the repair company is represented by a TADD[4]. As described in Section 4, deadlines for various activities are decided during contract negotiation between the parties. Deadlines are defined in terms of number of days. For example consider two contract clauses to be monitored:

- *If RC accepts a repair request it sends a proposal to C within 5 days* - clause (2) in table 1. A snippet of the TADD for the clause is shown in the Figure 4.

- *If C sends a damaged vehicle to RC, it assess the damage to the vehicle within 3 days* - clause (3) in table 1. A snippet of the TADD for the clause is shown in the Figure 5.

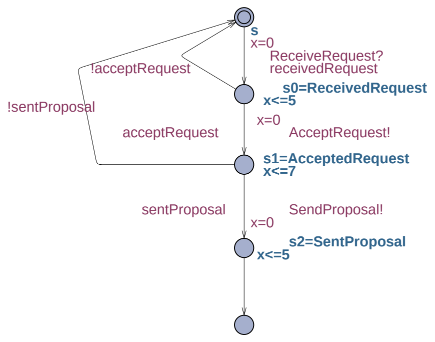

Figure 4. TA specification of clause (2)

Figure 6 describes for clause (2) the timeline in number of days, status of RC in terms of tasks executed, snapshots taken by the logger and sent to the RSA, snapshots sent to the monitoring engine by the RSA and the results from monitoring. Here, x denotes the clock against which deadlines are monitored. Since deadlines for this contract is in days, the tick for clock update is defined to be 1 day.

[4]The complete TADD for the example being too large, we do not include it in the paper

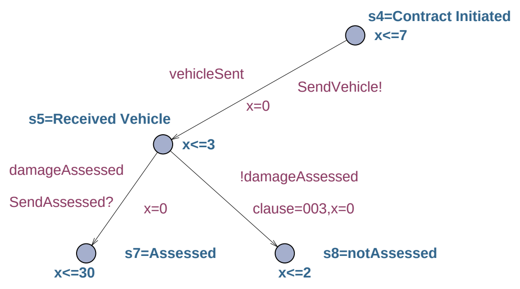

Figure 5. TA specification of clause (3)

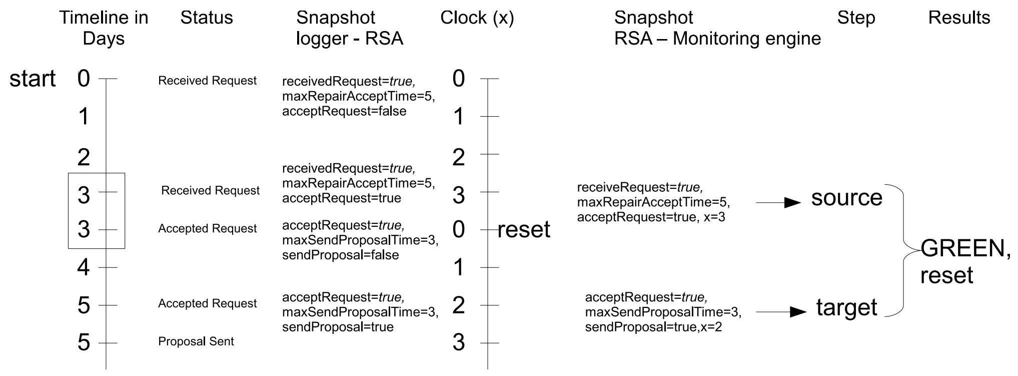
Timeline in Days
Status
Snapshot logger - RSA
Clock (x)
Snapshot RSA – Monitoring engine
Step
Results
start
0
1
2
3
3
4
5
5
Received Request
receivedRequest=true, maxRepairAcceptTime=5, acceptRequest=false
0
1
2
Received Request
receivedRequest=true, maxRepairAcceptTime=5, acceptRequest=true
3
receiveRequest=true, maxRepairAcceptTime=5, acceptRequest=true, x=3
source
Accepted Request
acceptRequest=true, maxSendProposalTime=3, sendProposal=false
0
reset
1
GREEN, reset
Accepted Request
acceptRequest=true, maxSendProposalTime=3, sendProposal=true
2
acceptRequest=true, maxSendProposalTime=3, sendProposal=true,x=2
target
Proposal Sent
3

Figure 6. Runtime valuations for clause (2)

The first snapshot is passed to RSA at $x = 0$ from the logger, when a request for repair is received. The request is accepted at $x = 3$ by the service and a new snapshot is passed by the logger. The clock x is reset as part of the TA specification. As per the contract, once a request has been accepted, the repair proposal has to be sent within 9 days. When RC accepts the proposal, a snapshot is again sent by the logger to the RSA at $x = 2$. The snapshot taken at $x = 3$, before reset and at $x = 2$ after reset are sent by the RSA as a pair - or as a "step" to RMCS. The results returned by the monitoring engine are $\{GREEN, reset\}$. $GREEN$ signifies that the step is a valid step, i.e., a valid transition and $reset$ indicates that the clock has been reset. Execution steps are evaluated for valid transitions as per the methodology described in Section 2.

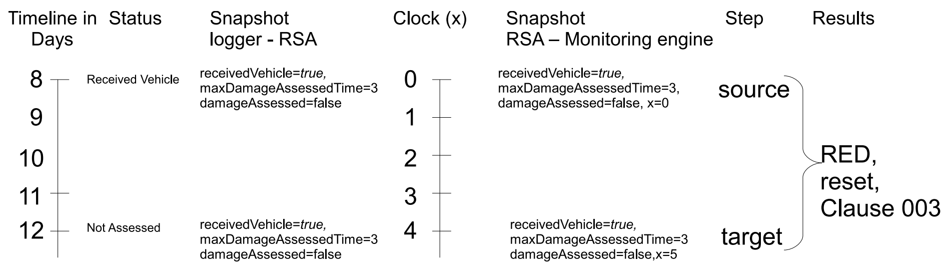
Timeline in Days
Status
Snapshot logger - RSA
Clock (x)
Snapshot RSA – Monitoring engine
Step
Results
8
9
10
11
12
Received Vehicle
receivedVehicle=true, maxDamageAssessedTime=3 damageAssessed=false
0
1
2
3
4
receivedVehicle=true, maxDamageAssessedTime=3, damageAssessed=false, x=0
source
RED, reset, Clause 003
Not Assessed
receivedVehicle=true, maxDamageAssessedTime=3 damageAssessed=false
receivedVehicle=true, maxDamageAssessedTime=3 damageAssessed=false,x=5
target

Figure 7. Runtime valuations for clause (3)

Step	state	Explanation
step 1	source	RC waits to receives the request for repairing cars.
	target	RC receives the request for repairing x cars. In table 3 we present an example of the clock and variable valuations for three cars.
step 3	source	RC accepts the request for repairing x cars.
	target	RC sends repair proposals for repairing x cars.

Table 2. Explanation of trace contents for steps 1 and 3

step nr	nr of cars	nr of int variables	nr of clocks	Nc/Nvars	time [s]	answer
1	10	10	10	6779/16528	<1	YES
	20	20	20	17738/43455	<1	
	300	300	300	265741/652852	4.3	
3	10	10	10	6743/16431	<1	NO
	30	30	30	26781/65822	<1	
	300	300	300	265811/653052	5.4	

Table 3. The experimental results. Parameters of the example are described in the text; size of encoding: $Nc/Nvars$ is the number of clauses/Boolean variables in the result CNF formula; time refers to checking this formula using the tool Minisat.

Figure 7 describes for clause (3) the timeline in number of days, a snapshot passed to RSA at $x = 0$ from the logger when a vehicle for repair arrives, snapshots sent to the monitoring engine by the RSA and the results from monitoring. As per the contract, once a damaged vehicle has arrived the damage has to be assessed within 3 days. A snapshot is again sent by the logger to the RSA at $x = 5$. The snapshot taken at $x = 0$ and at $x = 5$ are sent by the RSA as a pair - or as a “step” to RMCS. The results returned by the monitoring engine are $\{RED, reset, 003\}$. RED signifies that a violation has occurred, i.e., the damage was not assessed within the deadline, $reset$ indicates that the clock has been reset and 003 indicates the clause index that has been violated.

4.2. Experimental results and Discussion

In order to validate our methodology, we implemented the above case study and monitored several run-time execution steps for the service. To provide an indication of the number of variables the toolkit can monitor at the same time we scaled the example described above parametrizing the number of cars in the contract. As one clock and one integer variable are associated with every car, numbers of clocks and int variables grow respectively. Notice that the bigger the values these vars can have, the more bits are needed for encoding them.

We scaled the example above so that the client is now interested in getting x cars repaired. The request for all these repairs is included as a single contract.

Table 2 explains the contents of traces for contract clauses 1 and 3 (see Table 1). Table 3 presents experimental results. It can be stated that the approach performs extremely well against explicit approaches, which, although more immediate in their construction, typically fail to scale due to their memory footprint. This phenomenon could be even more visible if we could have a network of automata instead of a single automaton defining a contract. The experiments show the approach can monitor effectively several hundreds of variables. This is sufficient for very complex monitoring of key aspects of a service. We did not optimise the monitoring process in any way; we expect our results to improve significantly by tailoring the approach to a particular problem we wish to monitor. Indeed, observe that the methodology above could be parallelised over several engines on top of the web service with each engine monitoring different independent contracts or clauses in a contract.

As can be seen from the tables above, we found the only time consuming step of our methodology to be the construction of the automaton representing all behaviours. However this only needs to be done once, tools to assist the user in the design exist, and it can then be used for all monitoring purposes. Additionally it is to be noted that for complex applications, a representation of the service composition in an automata-based framework (or something equivalent) is expected to be produced during the design phase, so the construction above may in practice be derivable from existing formalisations of the composition under analysis.

5. Related work and conclusions

In this paper we presented a symbolic approach based on timed automata for the runtime monitoring of contract regulated agent based WS. Several previous efforts have investigated various formalisms and frameworks for the monitoring of functional and non-functional properties of services. Within the multi-agent community, Modgil et al [18] present a somewhat similar approach, where norms defining compliance or violation are specified as augmented transition networks. The monitoring technique adopted here is corrective, whereas we propose a *predictive* approach where agents could be warned if one of the next states on transition would be a red state. An alternative approach is presented in [11] where *overhearing* is used as a monitoring technique. In contrast, the agents in our system explicitly communicate their state to the monitoring engine.

In [20] the authors propose an approach based on Aspect Oriented Programming. The methodology is based on QoS requirements and does not consider complex contract like constraints. The monitoring problem has also been considered for several formalisms in papers [25, 2, 4, 23, 19, 16, 14, 3]. Table 4 presents a brief summary.

Timed automata have been used in earlier work such as [13] on monitoring and fault diagnosis of systems, while [24] presents an approach which also uses timed automata for monitoring SLAs. The aims of the above approaches are however quite different from our objectives in this paper. However [13, 24] are not concerned with local monitoring of contract-based executions.

Further none of the approaches above is based on a symbolic technique, which as shown in this paper offers a significant performance advantage. This is due to the fact that, differently from explicit approaches, in our framework histories and pending contracts are not stored in memory during the monitoring. This positively impacts the scalability of the approach and is particularly useful when monitoring

	Properties	**Monitoring spec**	**Web service spec**
[25]	general	ITL-formulae	OWL-S
[2]	boolean, time-related and statistic properties	RTML	BPEL, java
[4]	general	Algebraic specification	BPEL
[23]	protocols	Automata, EaGLe	-
[19]	rights and obligation	FSMs	B2B Object middleware
[16]	general	Event calculus	BPEL
[14]	interaction constraints	FSAs	OWL-S
[3]	timeouts, external errors, contracts	Assertions languages	BPEL, C#

Table 4. Summary of approaches

multiple and long running contracts between several services. As a case study we presented the monitoring of contracts for a repair company. Although the TADD for the service is not large enough to exploit the full capabilities of RMCWS, we believe it is still sufficiently significant to demonstrate the methodology and scope of the proposed approach. Experiments demonstrate that larger scenarios would be handled just as well by the technique.

While verification is still an aspect of systems validation we are not aware of symbolic attempts to the runtime monitoring of these notions. It seems to us that it may be of interest to investigate whether this could be achieved in ways related to the technique presented here.

Much work remains to be done. An important part of our future work is the translation to TADDs from high level specification standards such as WSBPEL. Developing such a translation is non trivial as most standards do not support the explicit representation of timing constraints on prescribed activities. These standards therefore need to be augmented with such support. Additionally, we are interested in developing an interactive compiler for services specified in WSBPEL to be compiled into our TADD representation.

References

[1] R. Alur. Timed Automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 63–71, 2006.

[3] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 193–202. ACM, 2004.

[4] D Bianculli and C. Ghezzi. Monitoring conversational web services. In *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*, pages 15–21. ACM, 2007.

[5] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS: A tool for verifying Timed Automata and Estelle specifications. In *TACAS'03: Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 278–283. Springer-Verlag, 2003.

[6] N. Desai, N. C. Narendra, and M. P. Singh. Checking correctness of business contracts via commitments. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 787–794, 2008.

[7] N. Eén and N. Sörensson. MiniSat. http://minisat.se/MiniSat.html.

[8] Andrew D. H. Farrell, Marek J. Sergot, Mathias Sallé, and Claudio Bartolini. Using the event calculus for tracking the normative state of contracts. *Int. J. Cooperative Inf. Syst.*, 14(2-3):99–129, 2005.

[9] N. Fornara and M. Colombetti. Specifying and enforcing norms in artificial institutions. In *AAMAS'08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1481–1484, 2008.

[10] G. J. Holzmann. *SPIN Model Checker, The: Primer and Reference Manual*. Addison Wesley Professional, 2003.

[11] Gal A. Kaminka, David V. Pynadath Milind Tambe, David V. Pynadath, and Milind Tambe. Monitoring teams by overhearing: A multi-agent plan-recognition approach. *Journal of Artificial Intelligence Research*, 17:2002, 2002.

[12] A Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *J. Netw. Syst. Manage. 11(1)*, pages 257–265, 2003.

[13] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *SPIN'04: the 11th International SPIN Workshop on Model Checking of Software*, LNCS, pages 109–126, 2004.

[14] Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *ASWEC '06: Proceedings of the Australian Software Engineering Conference (ASWEC'06)*, pages 70–79. IEEE Computer Society, 2006.

[15] A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.

[16] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: initial implementation and evaluation experience. In *ICWS'05, IEEE International Conference on Web Services*, pages 257–265, 2005.

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] S. Modgil, N. Faci, F. Meneguzzi, N. Oren, S. Miles, and M. Luck. A framework for monitoring agent-based normative systems. In *AAMAS '09: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 153–160, 2009.

[19] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne. Contract representation for run-time monitoring and enforcement. *CEC*, pages 103–110, 2003.

[20] S. Dustdar O. Moser, F. Rosenberg. Non-intrusive monitoring and service adaptation for ws-bpel. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 815–824. ACM, 2008.

[21] OASIS Web service Business Process Execution Language (WSBPEL) TC. Web service Business Process Execution Language Version 2.0, 2007.

[22] P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[23] Marco Pistore, F. Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.

[24] F. Raimondi, J. Skene, L. Chen, and W. Emmerich. ”efficient monitoring of web service slas”. Technical report, UCL, London, 2007.

[25] Monika Solanki. *A Compositional Framework for the Specification, Verification and Runtime Validation of Reactive Web Service*. PhD thesis, De Montfort University, Leicester, UK, October 2005.

[26] M. Wooldridge. *An introduction to multi-agent systems*. John Wiley, England, 2002.

[27] A. Zbrzezny and A. Półrola. SAT-based reachability checking for timed automata with discrete data. *Fundamenta Informaticae*, 79(3-4):579–593, 2007.