

# Automatic Generation of Real-Time Observers for Monitoring Web Services

Moez Krichen  
ReDCAD Research Unit  
DGIMA, ENIS,  
B.P. W 1173, Sfax, Tunisia.  
moez.krichen@redcad.org

Monika Solanki  
Imperial College London,  
London, SW7 2AZ,  
United Kingdom.  
m.solanki@imperial.ac.uk

## ABSTRACT

We explore the use of the timed test generator tool TTG as an automatic generator of observers for monitoring Web services. Our starting point is a service behaviour specified as a network of Timed Automata written in IF language. From the latter an observer is automatically synthesized. The observer checks whether a sequence of observations conform to the specification. We applied our method on some non conforming traces generated from a holiday booking web service example. The non conformance was detected using continuous time semantics.

## 1. INTRODUCTION

In this paper we study the problem of monitoring real time Web services. Monitoring of runtime service behaviour has several objectives, from checking if the functionality has been implemented correctly with respect to a given specification, to measuring performance and quality-of-service parameters. There has been an ongoing interest in the research community to develop techniques that bridge the gap between formal verification and testing. Runtime verification or monitoring assures the correctness of software execution at runtime with respect to a formal requirement specification. While other verification techniques, such as testing, model checking, and theorem proving, aim to ensure universal correctness of programs, the intention of runtime monitoring is to determine whether the current execution preserves specified properties and identifies possible unwanted behaviours.

In this paper our objective is to automatically synthesise a monitor, that observes a trace of web service behaviour generated at runtime and infers whether the service satisfies a certain specification. The service to be monitored is a black box i.e. no implementation details of the service are available. Our methodology relies on recording the observable behaviour generated by the service at runtime and checking conformance against a given specification.

We propose the use of TTG tool [10] for monitoring of web services. TTG tool is a timed test generator. It is built on top of the IF environment [6]. The IF modeling language allows to specify systems consisting of many processes communicating through message passing or shared variables and includes features such as hier-

archy, priorities, dynamic creation and complex data types. These features of the language make it suitable for modelling services for monitoring.

We are inspired by the methodology presented in [4]. Our technique is illustrated in Figure 2. It consists of the following phases:

1. Deriving a timed-automaton specification from the service description.
2. Automatic generation of an observer from the timed-automaton specification.
3. Instrumentation of the service.
4. Execution and testing of the instrumented service.

In the figure, solid arrows represent model and program transformations and dashed arrows represent data flow (output/input). We elaborate on each of the above phases in detail in section 4. We illustrate our methodology by monitoring the behaviour of a holiday booking service and report on experimental results.

The remaining part of the paper is structured as follows. Section 2 defines the timed automaton model. Section 3 describes the holiday booking case study. Section 4 discusses our methodology in detail. Section 5 gives a brief introduction on the usage of the timed test generator TTG. Section 6 summarizes the obtained results. Section 7 discusses the related work. Finally, Section 8 concludes the paper and gives directions for future work.

## 2. TIMED AUTOMATA

We use timed automata [1] with deadlines.

DEFINITION 1. A timed automaton over Act is a tuple  $A = (Q, q_0, X, \text{Act}, E)$ , where:

- $Q$  is a finite set of locations.
- $q_0 \in Q$  is the initial location.
- $X$  is a finite set of clocks.
- $E$  is a finite set of edges.

Each edge is a tuple  $(q, q', \psi, r, d, a)$ , where:

- $q, q' \in Q$  are the source and destination locations.
- $\psi$  is the guard, a conjunction of constraints of the form  $x \# c$ , where  $x \in X$ ,  $c$  is an integer constant and  $\# \in \{<, \leq, =, \geq, >\}$ .
- $r \subseteq X$  is a set of clocks to reset to zero.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

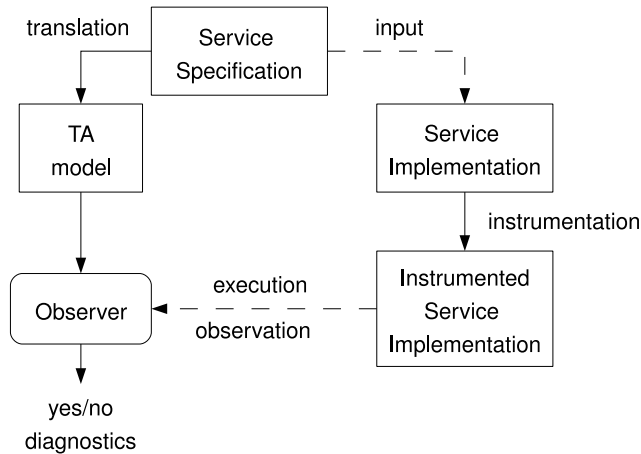


Figure 1: Monitoring Architecture.

- $d \in \{\text{lazy, delayable, eager}\}$  is the *deadline*.
- $a \in \text{Act}$  is the *action*.

A timed automaton  $A$  defines an infinite *timed labeled transition system* (TLTS) which is denoted  $L_A$ . Its states are pairs  $s = (q, v)$ , where  $q \in Q$  and  $v : X \rightarrow \mathbb{R}$  is a clock *valuation*.  $\vec{0}$  is the valuation assigning 0 to every clock of  $A$ .  $S_A$  is the set of all states and  $s_0^A = (q_0, \vec{0})$  is the initial state.

Discrete transitions are of the form  $(q, v) \xrightarrow{a} (q', v')$ , where  $a \in \text{Act}$  and there is an edge  $(q, q', \psi, r, d, a)$ , such that  $v$  satisfies  $\psi$  and  $v'$  is obtained by resetting to zero all clocks in  $r$  and leaving the others unchanged. Timed transitions are of the form  $(q, v) \xrightarrow{t} (q, v + t)$ , where  $t \in \mathbb{R}, t > 0$  and there is no edge  $(q, q'', \psi, r, d, a)$ , such that: either  $d = \text{delayable}$  and there exist  $0 \leq t_1 < t_2 \leq t$  such that  $v + t_1 \models \psi$  and  $v + t_2 \not\models \psi$ ; or  $d = \text{eager}$  and  $v \models \psi$ .

### 3. CASE STUDY: A HOLIDAY BOOKING WEB SERVICE

In this section, we present as our case study, the simplified version of a holiday booking service. The example is a composition of the client ( $CL$ ), the hotel booking service ( $HB$ ) and a car hire service ( $CH$ ). Car hire is a complementary service provided by the hotel to its clients.

The sequence of messages exchanged between the services is illustrated in Figure 2. The interaction is initiated by the  $CL$  sending a room availability request to  $HB$ . If  $HB$  confirms the availability,  $CL$  proceeds with the booking. On confirmation,  $HB$  books a complimentary car through  $CH$  for  $CL$ . For a room to be reserved, the  $CL$  has to pay an upfront fee. The fees are refundable in case of a cancellation. The refunds  $CL$  can expect are defined as part of the booking terms and conditions. The interactions between the services are governed by these conditions as well as the arrangements between  $CH$  and  $HB$ .

In this paper our objective is to monitor the behaviour of  $HB$ , under constraints defined for the interactions. We are specifically interested in the scenario, when a confirmed booking from the  $CL$  is cancelled. We outline below some of the key properties to be encoded as part of the behaviour specification of  $HB$  for such a scenario.

- If a booking request is received, it has to be replied within 3 days.
- If a cancellation message is received 24 hrs before check-in, refund paid is 80% of the booking fee.
- If the message is received between 24 and 12 hrs refund paid is 50% of the fee. In all other cases between 12 hrs and scheduled check in time, refund paid is 30% of the fee.
- In case of a no show, the  $CL$  is not refunded.
- In case of a cancellation,  $HB$  pays 5% of the fee to  $CH$  as compensation. The compensation has to be paid before the  $CL$  is refunded.
- After making a cancellation request, the  $CL$  has to wait between 2 to 5 weeks for the refund, if any, to be processed.

In the following sections, we show how the behaviour of the composed system can be monitored using TTG.

### 4. METHODOLOGY

The first step is to derive a timed automaton, or a network of timed automata (TA) from the service description. In the web service domain, WSBPEL [13] and OWL-S [17], are some popular and widely used standards for describing service behaviour, their composition and interaction protocols. Due to space restrictions, we do not discuss the issues of mapping between these standards and the IF language, and focus on the non trivial monitoring technique as presented below.

Having obtained the TA specification  $A$ , the next step consists in generating automatically an *observer* for  $A$ . The observer is a monitoring device. It observes the service and checks whether the trace generated by the service conforms to  $A$ . The observed traces are sequences of observable events and associated time-stamps. The accuracy of the time-stamps depends on the accuracy of the clocks of the observer.

There are mainly two types of observers (we follow the terminology of [9]). *Analog* observers, which can observe real-time precisely, and *digital* observers, which measure time with a clock ticking at a given period.

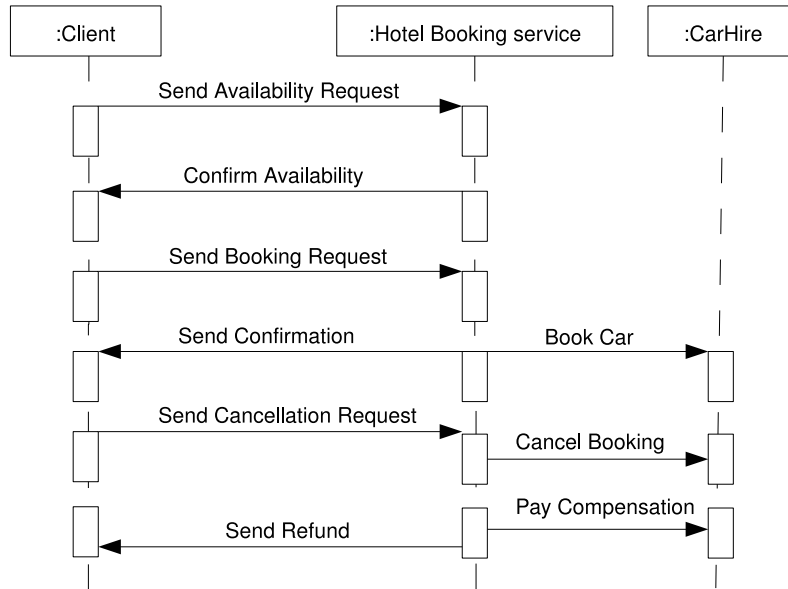


Figure 2: Interaction of messages between client and holiday booking service

Digital-clock observers are clearly more realistic to implement, since in practice the observer will only have access to a finite-precision clock. However, analog-clock observers are still useful, for instance, when the implementation is discrete-time but its time step is not known a-priori. In this work, we restrict ourselves to the case of digital observers.

An observed trace conforms to  $A$  if it can possibly be generated by  $A$ . Notice that  $A$  is typically modeled as a network of timed automata, which induces non-determinism and internal communication between the automata. These are artifacts of the model, irrelevant to the external behavior and to the specification itself. Thus, we “hide” them, by considering them as unobservable events. This means that the observer checks if the observed trace is a possible observation resulting from some trace of  $A$ .

The third step is the instrumentation of the service. It aims at interfacing the latter with the testing device (the observer). Two possibilities exist here. Either testing is performed *on-the-fly* (or *on-line*), that is, during execution of the service, which is connected to the observer at real-time. Or it is performed *off-line*, that is, by first executing the service multiple times to obtain a set of *log-traces*, then feeding these traces to the observer.

In both cases, the service must be able to expose a set of observable events to the observer. In the case of testing off-line, the service must also record the time-stamps of these events. For testing on-line, time-stamping can be done by the service or by the observer. In the latter case, possible interfacing delays must be taken into account.

Instrumentation can be done manually or automatically. Depending on the complexity of the service, it can be a non-trivial task. Care should be taken so that the instrumentation does not itself alter the behavior of the service. For instance, the overhead of added code should be minimal, so as not to affect execution times of the tasks in the service. These are problems inherent in any instrumentation process, and are beyond the scope of this work.

The final step is the testing procedure per-se. The traces generated by the service are fed to the observer, either in real-time (for on-the-fly testing) or off-line. The observer checks conformance

of each trace. If a trace is found non-conforming to the specification, the service is non-conforming. Otherwise, no conclusion can be made. However, confidence to the correctness of the service is increased with the number of tests.

## 5. TIMED TEST GENERATOR: TTG

The TTG tool may be used for both test generation and monitoring purposes. Four generation modes are possible:

1. *Interactive*: the user guides the test generation algorithm, resolving the non deterministic points (whether to issue an output or wait for an input, which output if many are possible, when to stop generating the test, etc.).
2. *Random*: the non-deterministic points are resolved randomly.
3. *Exhaustive*: all possible tests are generated up to a user-defined depth.
4. *Coverage*: a set of tests that achieves a user-defined coverage criterion is generated.

Test generation is beyond the scope of this work. More details about the test generation mode are to be found in [11].

TTG is written in C++. It works on linux iX86 platforms. It is built on top of the IF environment [6]. The IF modeling language allows to specify systems consisting of many processes communicating through message passing or shared variables and includes features such as hierarchy, priorities, dynamic creation and complex data types.

The IF tool-suite includes a simulator, a model checker and a connection to the untimed test generator TGV [8]. TTG is implemented independently from TGV. TTG uses the basic libraries of IF for parsing and symbolic reachability of timed automata with deadlines. For using TTG, one shall proceed as follows:

- Write your specification<sup>1</sup> in the IF syntax and save into a text file with “.if” extension (e.g., “spec.if”).

<sup>1</sup>specification = model of the SUT + model of the digital clock of the tester + priority rules.

- Run command “`runttg.sh spec.if`” (or “`runttg.sh spec`” for short) to generate the “TTG” executable file.
- Run the executable file “TTG”.

## 6. RESULTS

In this section, we present the results of monitoring the holiday web service using TTG. The behaviour of the hotel and client is illustrated in Figure 3 and Figure 4 respectively.

### 6.1 The IF model

We modelled our case study in IF language. The model is made up of three processes: the hotel process (Figure 3), the client process (Figure 4) and the digital-clock process (Figure 5).

The IF code for the hotel process has two states “1” and “2”. State “1” (the initial state) is when the hotel is not fully booked and state “2” is when it is fully booked. The process moves between these two states when it receives either the input-action “full?” or the input-action “not\_full?” from the environment.<sup>2</sup> When the hotel process occupies state “1” it can receive an input-action “check(id,d)” from the environment. The parameter “id” corresponds to the identifier of the new client which has sent the request. The parameter “d” corresponds to the number of days after which the client will check in. As soon as a request is received the hotel process creates a new client process “x” unless the current number of clients “nc” is greater or equal to “N”. The latter is the maximal number of client processes which can be handled by the hotel process. When a new client process is created the counter “nc” is incremented by one. Either at state “1” or state “2” when a “kill(x)” request is received the hotel process kills the process “x” and decrements “nc” by one.

The IF code of the clock process has only one state “1”. It models a digital periodic clock which sends a “tick” action after each one time unit. The elapsed time is measured using the continuous variable “tc”. A “tick” happens as soon as the guard “tc=12” becomes true.<sup>3</sup> The variable “tc” is reset to zero when a “tick” happens. The time unit considered in our example is equal to twelve hours.

The IF code of the client being too lengthy, we do not include it in the paper. It is made of 18 states. State “1” is the initial state. The process has three clocks “rt”, “dt” and “rft”. The “rt” clock guarantees a “confirm(id)” action to be sent at most three days after a “book(id)” request is received. Then immediately after confirmation a car booking request “book\_c(id)” is sent. The clock “dt” measures the time remaining before checking in. It defines four periods:

- “ $dt < -24$ ”: while this guard is true the process will be occupying state “4” until a “cancel(id)” action is observed or until the guard is no longer true due to time advance. If “cancel(id)” is observed an 80% refund will be paid back.
- “ $-24 \leq dt < -12$ ”: for this guard the process will be occupying state “5”. If “cancel(id)” is observed a 50% refund will be paid back.
- “ $-12 \leq dt < 0$ ”: if this guard is true the client process will be occupying state “9”. A “cancel(id)” action results in a 30% refund.

<sup>2</sup>The “?” and “!” marks are used to distinguish between input-actions and output-actions, respectively.

<sup>3</sup>We choose 12 since it is the smallest time constant appearing in the other processes.

- “ $dt=0$ ”: in this case the client will be occupying state “13”. Either action “no\_show(id)” or “check\_in(id)” will be observed then.

In the first three cases, before the refund is paid back to client the two actions “cancel\_c(id)” (cancelling the car booking) and “refund\_c(id)” (paying compensation to the car hire service) are observed consecutively.

Finally the clock “rft” guarantees that the refund is paid back within a period between two to five weeks at any case the “cancel(id)” action is observed. State “18” is a particular state. The process is destroyed as soon as this state is reached. Right before the destruction of the process a “kill(self)” signal is sent to the hotel process in order to update “nc” the current number of processes.

### 6.2 Monitoring properties

The method we propose is mainly based on the possibility of considering partial observability. The key idea is the following. We dispose of a model  $S$  of the functional behaviour of the service. We are also given a property  $\phi$  which has to be satisfied by the service. Two cases are then possible:

- Either  $S$  already satisfies  $\phi$ . In this case, we keep the model  $S$  as it is.
- Or  $\phi$  is not satisfied by  $S$ . In this case, we have to “improve” our model. We consider a new model  $S||\phi$ . That is the restriction of  $S$  satisfying  $\phi$ .

For simplicity, we assume that our model already satisfies the property in hand. Our goal is to monitor the implementation with respect to this property. For this purpose, we proceed as follows. First, we identify the set of (observable) events implicated in the property  $\phi$ . Call it  $Act_\phi$ . Second for each given (timed) trace  $\sigma$  to monitor, we project  $\sigma$  on the set of observable actions in  $Act_\phi$ . That is we erase the actions of  $\sigma$  which are not in  $Act_\phi$ . We obtain a new trace  $\sigma_\phi$ . Finally, we apply TTG on the obtained trace  $\sigma_\phi$  with respect to the set  $Act_\phi$ .

In this manner, we are guaranteed to satisfy both the property  $\phi$  and the functional behaviour of the service. Each time we are interested in a different property  $\psi$  and a different trace  $\beta$  we simply compute the corresponding set  $Act_\psi$  and trace  $\beta_\psi$  and apply TTG on them.

### 6.3 Experimental results

In this section we report on the obtained experimental results. We consider four properties to monitor:

P1 Checking the functional behaviour of one client: We first consider the following trace<sup>4</sup>:

```

trace1 = check(1,27)?
          book(1)!
          tick!3
          confirm(1)!
          tick!4
          cancel(1)!
          tick!36
          refund80(1)! .

```

The corresponding set of observable actions is

<sup>4</sup>We write “tick!<sup>2</sup>” instead of “tick! tick!”, “tick!<sup>3</sup>” instead of “tick! tick! tick!” and so on.

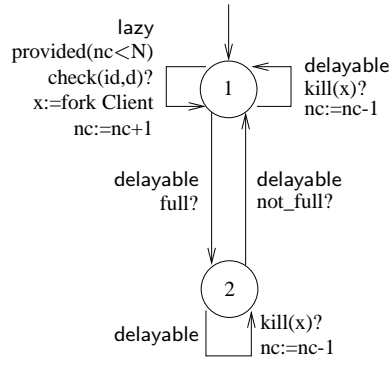


Figure 3: The hotel agent process.

```
obs1 = { tick! ,
          check(·,·)? ,
          book(·)! ,
          confirm(·)! ,
          cancel(·)! ,
          refund30(·)! ,
          refund50(·)! ,
          refund80(·)! }
```

Second we consider the trace “trace<sub>2</sub>” which is a slight modification of “trace<sub>1</sub>”. That is

```
trace2 = check(1,27)?
          book(1)!
          tick!3
          confirm(1)!
          tick!4
          cancel(1)!
          tick!36
          refund30(1)! .
```

The set of observable actions for this trace is “obs<sub>1</sub>” as well.

P2 Checking the functional behaviour of several clients: The trace we consider is

```
trace3 = check(1,27)?
          book(1)!
          tick!2
          check(2,26)?
          book(2)!
          tick!3
          confirm(1)!
          tick!
          confirm(2)!
          tick!4
          cancel(1)!
          tick!2
          cancel(2)!
          tick!6
          refund30(2)!
          tick!30
          refund80(1)! .
```

The corresponding set of observable actions is still “obs<sub>1</sub>”.

P3 Checking the hotel confirmation time-response: The trace to monitor is

```
trace4 = check(1,27)?
          book(1)!
          tick!2
          check(2,26)?
          book(2)!
          tick!3
          confirm(1)!
          tick!5
          confirm(2)!
```

The corresponding set of observable actions is

```
obs2 = { tick! ,
          check(·,·)? ,
          book(·)! ,
          confirm(·)! }
```

P4 Checking the hotel refunding time-response: The trace to monitor is

```
trace5 = check(1,27)?
          book(1)!
          tick!16
          cancel(1)!
          tick!16
          refund30(1)!
```

The corresponding set of observable actions is

```
obs3 = { tick! ,
          check(·,·)? ,
          book(·)! ,
          cancel(·)! ,
          refund30(·)! ,
          refund50(·)! ,
          refund80(·)! }
```

The obtained results for the all considered traces are shown in Table 1. The first colon gives the property to check; the second colon the trace to monitor; the third colon the set of observable actions; the fourth colon tells whether the considered trace is accepted or not; the fifth colon gives the execution time in milliseconds and the sixth colon the depth of the found error if so.

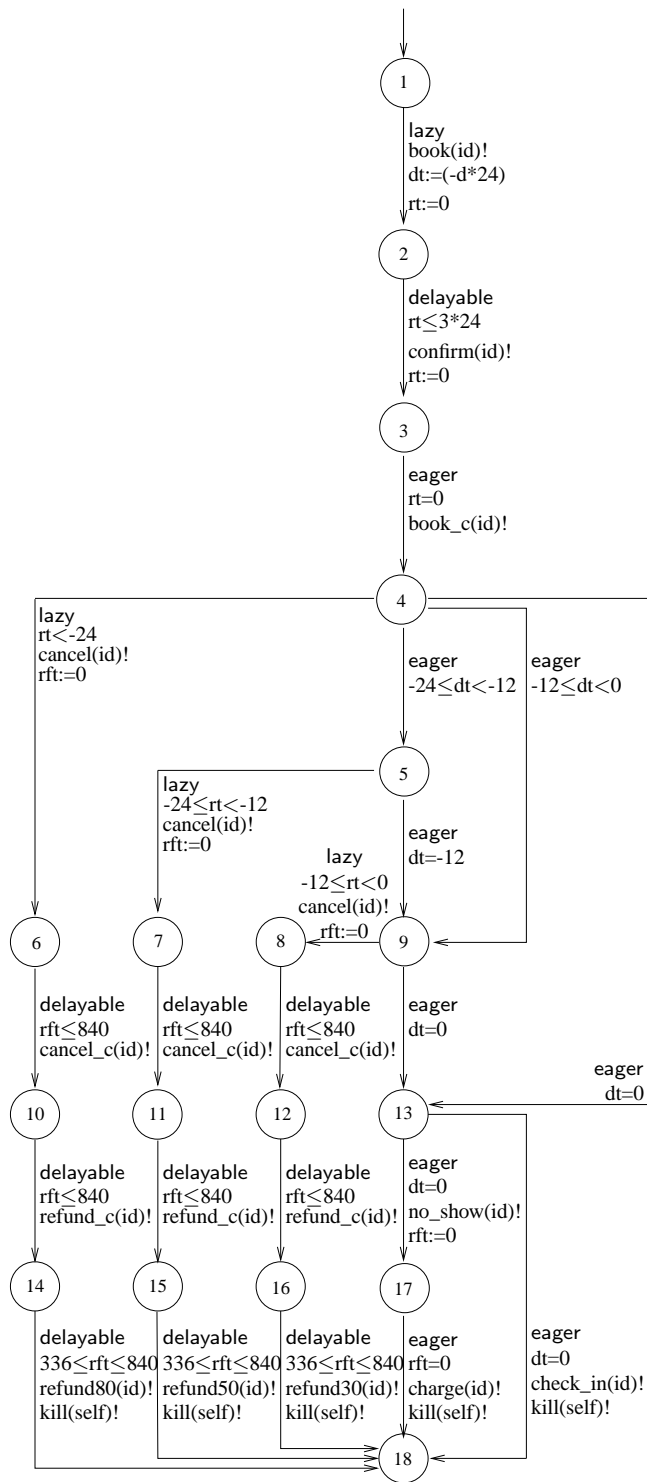
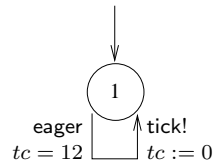


Figure 4: The travel agent process.



**Figure 5: The digital clock process.**

```

process Hotel(1);

  var nc nclts :=0;
  var x pid;
  var id clientid;
  var d clientdelay;

  state 1 # start ;

    deadline lazy;
    provided(nc < N);
    input check(id,d);
    x := fork Client(self);
    task nc := (nc + 1);
    task (Clientx).id := id;
    task (Clientx).d := d;
    nextstate 1;

    deadline eager;
    input kill(x);
    task nc := (nc - 1);
    nextstate 2;

    deadline eager;
    input full();
    nextstate 2;

  endstate;

  state 2 ;

    deadline eager;
    input kill(x);
    task nc := (nc - 1);
    nextstate 2;

    input not_full();
    nextstate 2;

  endstate;

endprocess;

```

**Figure 6: The IF code of the hotel process.**

```

process Clock(1);

    var tc clock;

    state 1 # start ;

        deadline eager;
        when tc=1;
        output tick();
        set tc:=0;
        nextstate 1;

    endstate;
endprocess;

```

**Figure 7: The IF code of the digital clock process.**

P	Trace to monitor	Observable actions	y/n	Time (ms)	Error depth
P1	trace <sub>1</sub>	obs <sub>1</sub>	yes	128	-
P1	trace <sub>2</sub>	obs <sub>1</sub>	no	144	48
P2	trace <sub>3</sub>	obs <sub>1</sub>	no	24	27
P3	trace <sub>4</sub>	obs <sub>2</sub>	no	8	14
P4	trace <sub>5</sub>	obs <sub>3</sub>	no	476	36

**Table 1: Summary of experimental results.**

## 7. RELATED WORK

Monitoring service behaviour has been an active area of research. Several efforts [16] [2] [5] [14] [12] [3] have investigated varied formalisms and frameworks for the monitoring of functional and non functional properties of services. Table 2 presents a brief summary.

Two approaches which also use timed automata for monitoring and are closely related to our methodology, are [15], [7]. The authors monitor SLAs for web services using TAs. Each constraint to be monitored is expressed first as a formula in TCTL and then compiled into a TA. SOAP messages exchanged by the services are time stamped. Violations of constraints is detected by checking the acceptance of a timed word by the automata. The advantage our approach has is that all possible behaviours of the service, including those to be monitored, are encoded as a single network of timed automata in the IF language. This also opens up the possibility of monitoring other behaviours of the service at a later stage, originally not considered critical.

## 8. CONCLUSION AND FUTURE WORK

In this paper we have shown, how automatic observers can be generated for monitoring real time web services using the timed test generator tool, TTG. The service behaviour is specified as a timed-automaton, compiled into the IF language.

The main contributions of this work are the following:

- We model the whole behaviour of the service. We assume that this model satisfies all properties of interest. The model is written in IF language. The latter allows to consider dynamic creation of processes.
- Monitoring a trace against a given property (satisfied by the model) is simply achieved by applying TTG with respect to the set of observable actions appearing within the considered property.
- We consider digital-clocks which are modeled as timed automata as well. More precisely, the model of the digital-clock is given as a separate IF process. The digital-clock may be periodic or not, deterministic or not, etc. Even though we use digital-clocks, we still consider the analog continuous semantics of the model of the system.

We are currently working on extending our monitoring approach to identify patterns of service failure. The basic idea is to provide a long term solution to service developers, whereby faults in service behaviour recorded over a period of time can be analysed and the service implementation suitably modified. We are also investigating the possibility of synthesizing “self healing” intelligent observers, for dealing with failures of services in the short term. Finally, we propose to report on the mapping between web service standards and the IF language in the near future.

## 9. REFERENCES

- [1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [2] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.
- [3] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202, New York, NY, USA, 2004. ACM.
- [4] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing conformance of real-time applications by automatic generation of observers. In *4th International Workshop on Runtime Verification (RV'04), Barcelona, Spain*, volume 113 of *ENTCS*, pages 23–43. Elsevier, 2005.
- [5] Domenico Bianculli and Carlo Ghezzi. Monitoring conversational web services. In *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*, pages 15–21, New York, NY, USA, 2007. ACM.
- [6] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: a validation environment for timed asynchronous systems. In *Computer Aided Verification, 12th International Conference (CAV'00), Chicago, IL, USA*, volume 1855 of *LNCS*, pages 543–547. Springer, 2000.
- [7] W. Emmerich F. Raimondi, J. Skene. Proceedings of provecs 2007- tools. 2007.
- [8] J.C. Fernandez, C. Jard, T. Jérón, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Computer Aided Verification, 8th International Conference (CAV'96), New Brunswick, NJ, USA*, volume 1102 of *LNCS*, pages 348–359. Springer, 1996.
- [9] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Automata, Languages and Programming, 19th International Colloquium (ICALP'92), Vienna, Austria*, volume 623 of *LNCS*, pages 545–558. Springer, 1992.
- [10] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *11th International SPIN Workshop on Model Checking of Software (SPIN'04), Barcelona, Spain*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004.
- [11] M. Krichen and S. Tripakis. An expressive and implementable formal framework for testing real-time systems. In *The 17th IFIP Intl. Conf. on Testing of Communicating Systems (TestCom'05), Montreal, Canada*, volume 3502 of *LNCS*, pages 209–225. Springer, 2005.
- [12] G. Mahbub, K.; Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: initial implementation and evaluation experience. In *IEEE International Conference on Web Services*, volume 1, pages 257–265. IEEE, July 2005.
- [13] OASIS Web service Business Process Execution Language (WSBPEL) TC. Web service Business Process Execution Language Version 2.0, 2007.
- [14] Marco Pistore, F. Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.
- [15] F. Raimondi, J. Skene, L. Chen, and W. Emmerich. Efficient monitoring of web service slas. Technical report, UCL, London, 2007.
- [16] 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.
- [17] The OWL-S Coalition. OWL-S 1.1 Release., 2004. <http://www.daml.org/services/owl-s/1.0/>.

Approach	Properties	Monitoring spec	Web service spec
[16]	general	ITL-formulae	OWL-S
[2]	boolean, time-related and statistic properties	RTML	BPEL, java
[5]	general	Algebraic specification	BPEL
[14]	protocols	Automata, EaGLE	-
[12]	general	Event calculus	BPEL
[3]	timeouts, external errors, contracts	Assertions languages	BPEL, C#
[15]	SLAs	TCTL & Timed automata	SOAP

**Table 2: Summary of monitoring approaches.**